Advanced *R* / *Bioconductor* Programming

Marc Carlson, Valerie Obenchain, Hervé Pagès, Paul Shannon, Dan Tenenbaum, Martin Morgan[1]

15-16 October 2012

[1]mtmorgan@fhcrc.org

Contents

Chapter 1

Introduction

The Advanced *R* / *Bioconductor* Programming workshop provides experienced *R* and *Bioconductor* users and package developers with an opportunity to develop advanced skills for creating performant, re-usable software. This course is relevant to *R* software development in general, but includes insights particularly relevant to development of bioinformatics. The material is structured around *R* packages and their implementation, including programming best practices, formal classes and methods, accessing data resources, strategies for measuring performance and managing large data, interfacing C code, and parallel evaluation. The course concludes with an extended tour of key *Bioconductor* packages for representation and manipulation of genomic data. Participants engage in lectures and hands-on exercises. Participants require a laptop with internet access and a current browser.

Dalgaard [4] provides an introduction to statistical analysis with *R*. Kabaloff [6] provides a broad survey of *R*. Matloff [7] introduces *R* programming concepts. Chambers [3] provides more advanced insights into *R*. Gentleman [5] emphasizes use of *R* for bioinformatic programming tasks. The *R* web site enumerates additional publications from the user community. The RStudio environment provides a nice, cross-platform environment for working in *R*.

Table 1.1: Tentative schedule.

Day 1		
	Morning	Orientation; *R* and *Bioconductor* Packages (package structure, name spaces, unit tests, documentation, version control).
	Afternoon	Formal Classes and Methods (S4 and reference classes). Accessing Data Base (sqlite) and Web Resources.
Day 2		
	Morning	Assessing Performance and Data Size. Calling C Code (.C and .Call interfaces).
	Afternoon	Parallel Evaluation. Extended Example: *IRanges*, *GenomicRanges*, *Biostrings* and friends.

Chapter 2

Packages

2.1 Anatomy of a package

2.1.1 Essentials: a minimal package

We start with a short *ad hoc R* function, one which proved useful in exploratory data analysis. If properly generalized, it may useful to others, so we decide to make it into a package.

The script loads a compendium of yeast expression data, and identifies which of 500 genes had highly correlated expression over 200 experimental conditions:

```
> correlationFinder <- function()
+ {
+     dataFile <- "sub_combined_complete_dataset_526G_198E.txt"
+     cor.threshold <- 0.85
+
+     tbl <- read.table(dataFile, sep='\t', header=TRUE, quote='',
+                       comment.char='', fill=TRUE, stringsAsFactors=FALSE)
+     rownames(tbl) <- tbl$X
+     exclude.these.columns <- !sapply(tbl, is, 'numeric')
+     if (any(exclude.these.columns))
+         tbl <- tbl[, !exclude.these.columns]
+     mtx.cor <- cor(t(as.matrix(tbl)), use='pairwise.complete.obs')
+     mtx.cor <- upper.tri(mtx.cor) * mtx.cor
+     max <- nrow(mtx.cor)
+
+     ret <- list()
+     for (r in seq_len(max)) {
+         zz <- mtx.cor [r,] > cor.threshold
+         if (any(zz)) {
+             ret[[ rownames(mtx.cor)[r] ]] <- rownames(mtx.cor)[zz]
+         } # if any
+     } # for r
+     ret
+ }
```

You may wish to get a copy of this function into *RStudio*. If so, follow these steps:

- From the Project menu, choose "New Project"
- If prompted, you may save (or not) your current workspace
- Click "Version Control"
- Click "Git"
- In the "Repository URL" box, paste `https://github.com/dtenenba/AdvancedR_stage1`
- Press the Tab key. The "Project Directory Name" box is automatically filled in.
- Click "Create Project"

R provides a function which helps us to create a fully-documented and easily shared package of code and data. It creates a directory structure, and populates it with an almost-working set of files. We will examine this directory structure, look at and make small modifications to these automatically generated files, build the package, and then `R CMD check` on it – a vital step when creating a package for distribution.

```
> package.skeleton('YeastmRNACor', code_files='yeastCorrelatedExpression.R')
```

These files and directories are created:

```
YeastmRNACor/Read-and-delete-me
YeastmRNACor/DESCRIPTION
YeastmRNACor/NAMESPACE
YeastmRNACor/man/correlationFinder.Rd
YeastmRNACor/man/YeastmRNACor-package.Rd
YeastmRNACor/R/yeastCorrelatedExpression.R
```

We will look at each of these, and addition files in Figure 2.1 in turn.

YeastmRNACor/Read-and-delete-me

1. Edit the help file skeletons in `man`, possibly combining help files for multiple functions.
2. Edit the exports in `NAMESPACE`, and add necessary imports.
3. Put any C/C++/Fortran code in `src`.
4. If you have compiled code, add a `useDynLib()` directive to `NAMESPACE`.
5. Run `R CMD build` to build the package tarball.
6. Run `R CMD check` to check the package tarball.

YeastmRNACor/DESCRIPTION

Package: YeastmRNACor
Type: Package
Title: Yeast Correlation Finder
Version: 0.99.0
Date: 2012-10-12
Author: Paul Shannon
Maintainer: Paul Shannon <pshannon@fhcrc.org>
Description: Find S.cerevisiae genes with correlated expression
License: Artistic-2.0

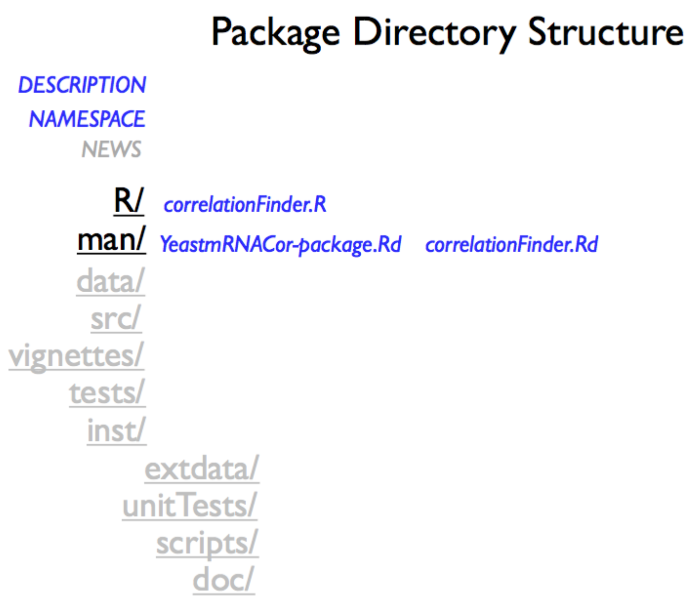


Figure 2.1: Package directory structure

YeastmRNACor/NAMESPACE

```
exportPattern("^[[:alpha:]]+")
```

YeastmRNACor/man/YeastmRNACor-package.Rd

```
\name{YeastmRNACor-package}
\alias{YeastmRNACor-package}
\alias{YeastmRNACor}
\docType{package}
\title{
  Yeast Correlation Finder
}
\description{
  Find S.cerevisiae genes with correlated expression
}
\details{
\tabular{ll}{
  Package: \tab YeastmRNACor\cr
  Type: \tab Package\cr
  Version: \tab 0.99.0\cr
  Date: \tab 2012-10-12\cr
  License: \tab Artistic-2.0\cr
  }
}
\author{
  Paul Shannon
  Maintainer: Paul Shannon <pshannon@fhcrc.org>
}
\references{
  Allocco et al, 2004, "Quantifying the relationship between
  co-expression, co-regulation and gene function":
}
\keyword{manip}
```

R documentation[1] provides a full list of the official keywords.

YeastmRNACor/man/correlationFinder.Rd

```
\name{correlationFinder}
\alias{correlationFinder}
\title{
correlationFinder
}
\description{
```

[1] http://svn.r-project.org/R/trunk/doc/KEYWORDS

```
  Finds yeast genes with correlated expression.
}
\usage{
  correlationFinder()
}
\details{
  Calculates the upper triangular correlation matrix from mRNA expression
  data; identifies genes whose expression is highly correlated.
}
\value{
  A named list, in which the names are genes, and the values are the
  genes highly correlated to each of them.
}
\author{
  Paul Shannon
}

\examples{
  \dontrun{
    correlated.list <- correlationFinder()
  }
}

\keyword{ array }
\keyword{ manip }
\keyword{ math }
```

YeastmRNACor/R/yeastCorrelatedExpression.R This file contains the original source code for our function.

2.1.2 A More Complete Package

package.skeleton created only two sub-directories, and just five files (see image above). A few more directories and files are needed to create a fully-compliant *Bioconductor* package, and a few more beyond that are sometimes needed as well. We will list and explain all of them here. The *MotifDb* package, to be examined later, will illustrate most of them.

data If your package provides data which the user will load and use directly, then the standard approach is to place a serialized (`xxx.Rdata`) file in the data directory. This file must then be documented as well, with a similarly named (`xxx.Rd`) man file. In other packages, data is provided only for package testing purposes, or the data is available to the user only through an interface, and in these cases the data files reside in inst/extdata, as we will discuss.

src If you have compiled code – typically C, C++, or Fortran – then the source files are placed here.

vignettes Vignettes are an essential tool, very helpful for introducing your package to users, and required by *Bioconductor*. They have an .Rnw suffix, and consist of commentary intermixed with executable code.

tests This is the traditional directory in which to place test code for your package. `R CMD check` automatically looks here. With the advent and popularity of the unitTest protocol, this directory contains just one file containing one line, which provides a hook to run the unitTests, described below.

inst By convention, the *R* package installer will place the contents of the `inst/` directory at the top level of the installed package.

inst/extdata As mentioned above, this directory contains data files which are used for unitTests and examples, or provided to the user after some processing. Files may be in a variety of formats, include text tab-delimited or yaml files, or serialized into Rdata. Data provided directly to the user of the package goes in the data directory.

inst/unitTests One or more unitTest files (discussed more fully below) can be placed here.

inst/doc Historically, vignettes files were place here. The vignettes directory is now preferred, but this directory is still supported.

inst/scripts Typically contains scripts used to create the package, for example, for parsing and transforming data which then ends up in the data directory, or in inst/extdata.

2.2 Version Control - Introduction

Version control is essential for:

- Saving your work
- Tracking the changes of a project
- Reverting to older versions
- Collaborating with others

Bioconductor uses Subversion, and *Bioconductor* package developers should learn the rudiments of that system. We are also intrigued by GitHub which provides an interesting model of distributed code development. Github is built on the Git version control system.

Bioconductor uses Subversion, and *Bioconductor* package developers should learn the rudiments of that system. We are also intrigued by GitHub which provides an interesting model of distributed code devlopment. Github is built on the Git version control system.

We'll introduce Github in the context of the package we've just started working on. Our original script is in this repository: `https://github.com/dtenenba/AdvancedR_stage1`. For now, just visit that URL with a web browser and look around. Notice that our original script is there, along with a data file.

The minimal package is in a different repository, `https://github.com/dtenenba/AdvancedR_stage2`. We can clone, or check out, check this repository, from within *RStudio* Server:

- From the Project menu, choose "New Project"
- If prompted, you may save (or not) your current workspace
- Click "Version Control"
- Click "Git"
- In the "Repository URL" box, paste `https://github.com/dtenenba/AdvancedR_stage2`
- Press the Tab key. The "Project Directory Name" box is automatically filled in.
- Click "Create Project"

The Github project is "cloned" into a directory called `AdvancedR_stage2`. Your current working directory is changed to this directory, both in the *R* console and in the File pane in the lower-right hand corner. Note that a Git pane appears in the pane at upper right. Those without *RStudio* can check out the repository at a command shell:

```
git clone https://github.com/dtenenba/AdvancedR_stage2
```

Note: Our use of version control in this course is a bit odd; We have several different repositories representing a package at different stages of its evolution. In real life, there would probably just be a single repository (though individual developers could create their own forks of it), and one could check out earlier iterations of the package.

2.3 Making the package more useful

Our package is great but it's of limited usefulness so far. It tries to open a file we may not have, and won't run on any other file we may have. And we can't change the correlation threshold. Let's fix that.

We'll make several changes:

- Put the data file in `inst/extdata`.
- Add a `dataFile` parameter to `correlationFinder()` with no default.
- Add a `cor.threshold` parameter to `correlationFinder()` with a default of 0.85.
- Update the man page to reflect these changes. Change the example so it works with the data file that's part of the package, (hint: `?system.file`) and remove the `dontrun` tag so that the example is actually run.
- Extra credit: Write a rudimentary vignette.
- Make sure the package passes `R CMD check` without warnings or errors. (Hint: use Tools/Shell to open a rudimentary command shell in *RStudio* Server).
- Install the package and view the man pages and vignette. Use `example()` to run the example in the man page.

Resources for this exercise:

- The Writing R Extensions Manual
- Source of *Bioconductor* Packages (log in with username and password 'readonly').

The package, with these changes incorporated, can be found at `https://github.com/dtenenba/AdvancedR_stage3`. Notice that it has a vignette. If a package has more than a couple of functions, a vignette is a must (and in fact is a requirement for *Bioconductor* packages). A package that does not have a vignette will have an automatically-generated reference manual, which is a compendium of all the man pages in the package, but that doesn't tell you which function to run first, or how to use the package for a given work flow. That's why vignettes are so critical, because as the name implies, they provide a narrative telling you how to use the package. The vignette in this package isn't very comprehensive, but it hints at some future directions in which the package could be taken.

2.4 Creating good packages and why it matters

2.4.1 Unit tests

We will follow the *Bioconductor* Unit Testing Guidelines page: `http://www.bioconductor.org/developers/unitTesting-guidelines`

2.4.2 Interoperability

When creating a new package it is useful to familiarize yourself with pre-existing classes and methods. Reusing the current infrastructure allows a new package to integrate smoothly with existing work flows. Additionally, the methods that have been written for these classes (i.e., subsetting, length, show, validity) are yours for free when you reuse or inherit from an existing class.

DESCRIPTION file Most fields in the DESCRIPTION file are self-explanatory. Here we touch on a few of the most important.

Description Posted on the package landing page. Often the first description of the package a user will see.
Depends Package is attached to search path but the namespace is not loaded.
Imports Package namespace is imported but the package is not attached to the search path.
Suggests Packages used in examples, tests or vignettes but not needed for current package functions.
biocViews These terms aid users in finding your package. For a list of terms see the *Bioconductor* web site[2].

NAMESPACE file The NAMESPACE file allows the user to control the package imports and exports. Importing allows the current package to make use of functions defined in other packages. Exporting enables the package author to expose their own functions as publicly available to other developers or kept private for internal use.

Through these directives the NAMESPACE file controls the search path where R looks for variables. First R looks inside the package namespace, then the imports, then the base namespace and then the normal search path.

Advantages of having a package namespace:

1. Your package will not be broken by functions defined by users in the global environment.
2. Your package will not be bothered by functions in other packages with the same name.
3. A namespace gives you the ability to clearly specify which functions are part of the public interface and which are private. R CMD check will not prompt you for documentation on non-exported functions, although it is often still useful to give them some documentation.

There are some disadvantages:

1. You have to maintain the NAMESPACE file
2. It is less convenient to debug/develop packages with a NAMESPACE because `R CMD INSTALL` must be run to be sure everything gets updated in the namespace properly. Although less convenient, you will be more certain that the behavior is really in your package and not a result of things hiding in your global environment.

Next we explore the *YeastmRNACor* package namespace. We start by demonstrating how easily we can break a package if imports are not properly defined in the `NAMESPACE` file. Load the *YeastmRNACor* package and then try redefining the `cor` function like this:

[2]http://www.bioconductor.org/packages/release/BiocViews.html#___Software

```
> cor <- function(...) cat("This is my version of cor\n")
```

Does the `correlationFinder` function still work?

The `cor` function is defined the *stats* package. This package is included with base R and is loaded when an R session is started. Start a fresh R session and type `sessionInfo()` to see what packages are loaded.

```
> sessionInfo()
```

Though the package is loaded, the namespace of *stats* has not been attached to our package namespace. The *stats* package is located after the `.GlobalEnv` on the search path. (`.GlobalEnv` is where we defined our new `cor` function.)

```
> search()
```

Add *stats* to the Imports field in the DESCRIPTION file and add the following line to the NAMESPACE.

```
import(stats)
```

Again try defining your own version of `cor` and verify that these changes protect against the redefinition of `cor`.

Importing and Exporting in the NAMESPACE `import` will import all functions from an existing package. Use `importMethodsFrom` and `importClassesFrom` if only a few functions or classes are needed.

Export generics and functions with `export`. Methods can be exported with `exportMethods` and classes with `exportClasses`.

Each exported function should have a man page with running examples.

The package, incorporating changes in this section, can be found at
https://github.com/dtenenba/AdvancedR_stage5.

2.4.3 From package to *Bioconductor* package

Bioconductor contributors should refer to the guidelines for submitted package, and the package submission page.

2.5 An Extended Example: *MotifDb*

2.5.1 Introduction

The MotifDb packages provides a collection of experimentally obtained protein-DNA binding sequence patterns accompanied by metadata. The `show(MotifDb)` method provides a summary:

```
> show(MotifDb)  # or simply:  MotifDb
| Created from downloaded public sources: 2012-Jul6
| 2086 position frequency matrices from 5 sources:
|    FlyFactorSurvey:  614
|               hPDI:  437
|        JASPAR_CORE:  459
|             ScerTF:  196
|           UniPROBE:  380
| 22 organism/s
|      Dmelanogaster:  739
|           Hsapiens:  505
|        Scerevisiae:  464
|          Mmusculus:  329
|        Rnorvegicus:    8
|           Celegans:    7
|              other:   34
Dmelanogaster-FlyFactorSurvey-ab_SANGER_10_FBgn0259750
Dmelanogaster-FlyFactorSurvey-ab_SOLEXA_5_FBgn0259750
Dmelanogaster-FlyFactorSurvey-Abd.A_FlyReg_FBgn0000014
Dmelanogaster-FlyFactorSurvey-Abd.B_FlyReg_FBgn0000015
Dmelanogaster-FlyFactorSurvey-AbdA_Cell_FBgn0000014
...
Mmusculus-UniPROBE-Zfp740.UP00022
Mmusculus-UniPROBE-Zic1.UP00102
Mmusculus-UniPROBE-Zic2.UP00057
Mmusculus-UniPROBE-Zic3.UP00006
Mmusculus-UniPROBE-Zscan4.UP00026
```

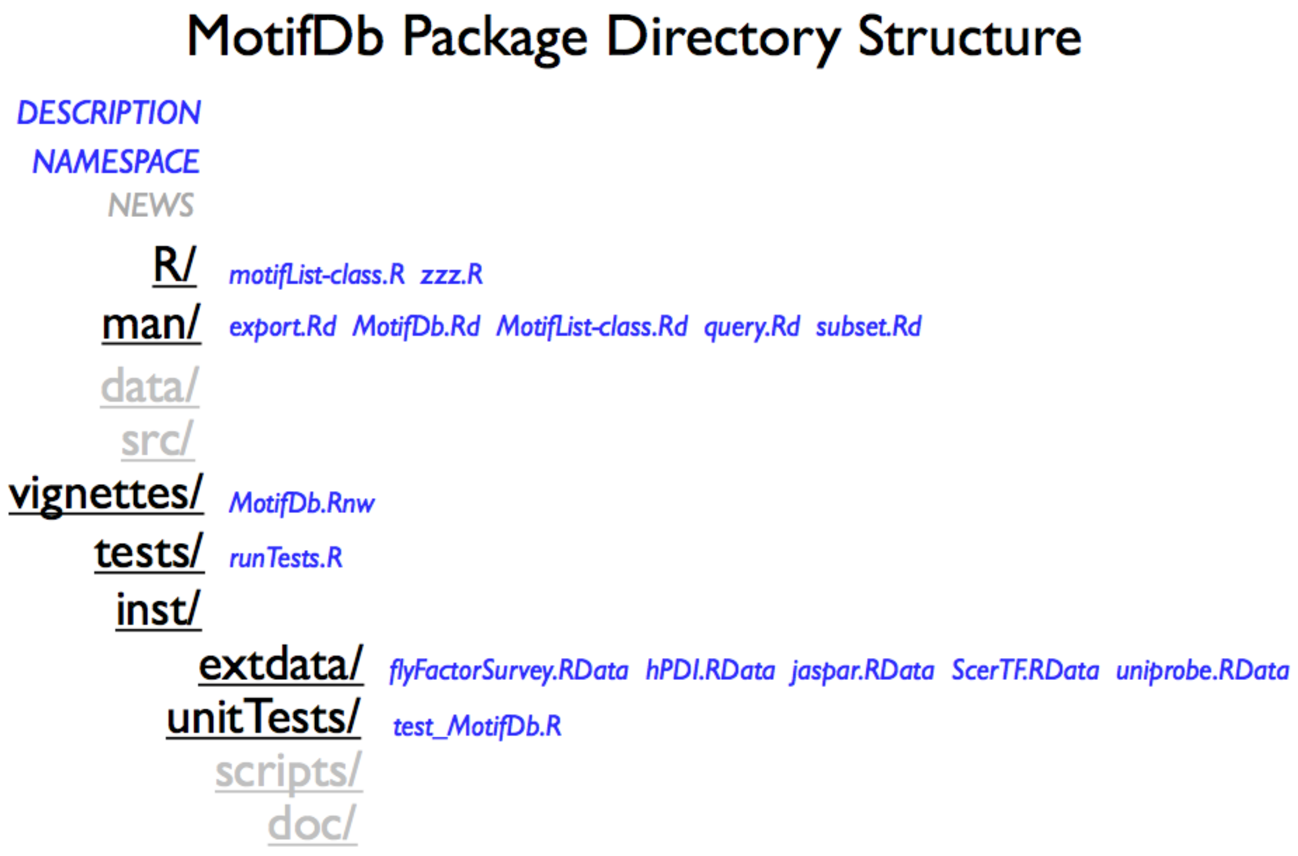
2.5.2 Highlights

The basic operations of MotifDb can be demonstrated with a few *R* commands, which provide us with the context for exploring the package structure and class design.

```
library(MotifDb)
library(seqLogo)
query(MotifDb, 'e2f3')
t(as.matrix (mcols (query(MotifDb, 'e2f3'))))
pfm <- query(MotifDb, 'e2f3')[[1]]
seqLogo(pfm)
```

2.5.3 Package structure

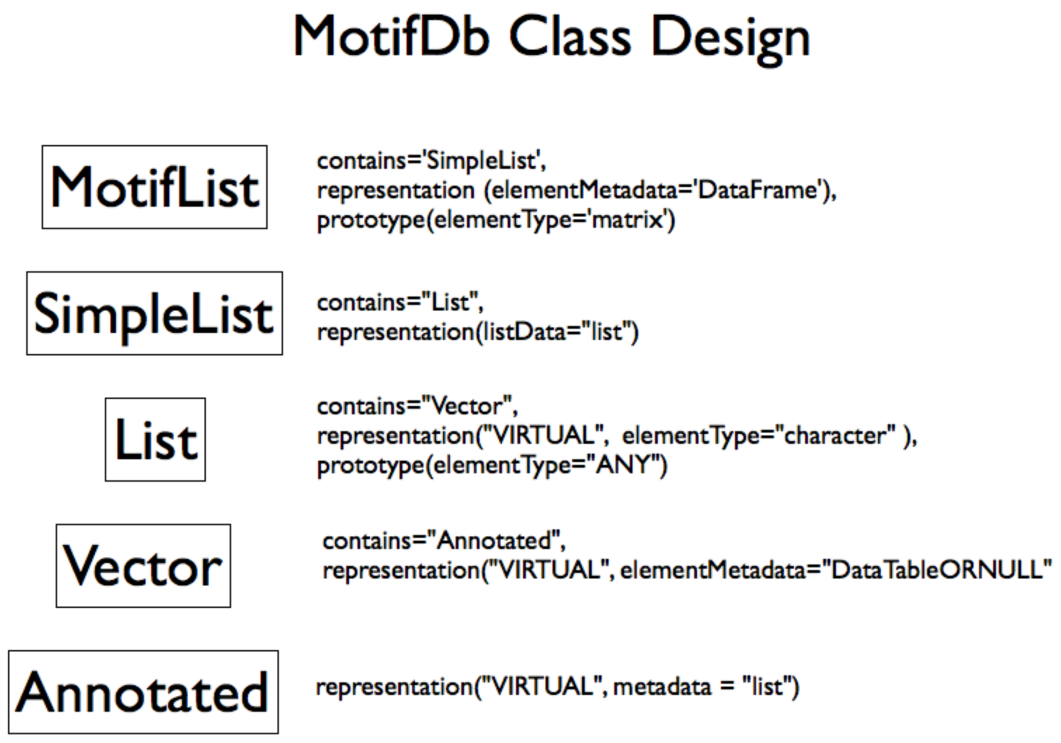
MotifDb directory structure is significantly richer than the first one we examimed (Section 2.1.1), but even so, some directories (src for compiled code, and data, for data directly provided to the user) are empty:


MotifDb Package Directory Structure
DESCRIPTION
NAMESPACE
NEWS
R/
motifList-class.R zzz.R
man/
export.Rd MotifDb.Rd MotifList-class.Rd query.Rd subset.Rd
data/
src/
vignettes/
MotifDb.Rnw
tests/
runTests.R
inst/
extdata/
flyFactorSurvey.RData hPDI.RData jaspar.RData ScerTF.RData uniprobe.RData
unitTests/
test_MotifDb.R
scripts/
doc/

2.5.4 Class Design

`MotifDb` is the name of the package; `MotifList` is the name of the class around which it is a built. When you load the class with `library(MotifDb)` an instance of `MotifList` is created, and populated with 2000+ matrices we have collected, along with their metadata.

However, the `MotifList` class is very simple. It contains only a few methods. Most of the capabilities it offers is accomplished with data structures and methods inherited from other *Bioconductor* infrastructure classes.



MotifDb Class Design

MotifList

contains='SimpleList',
representation (elementMetadata='DataFrame'),
prototype(elementType='matrix')

SimpleList

contains="List",
representation(listData="list")

List

contains="Vector",
representation("VIRTUAL", elementType="character"),
prototype(elementType="ANY")

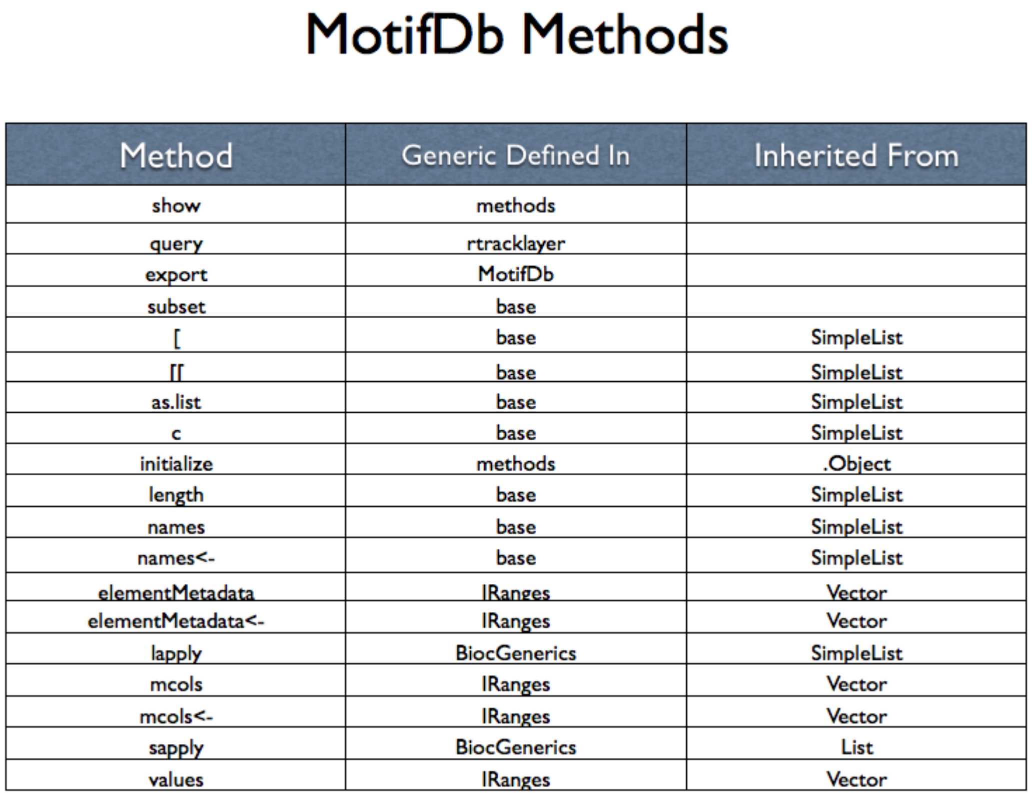
Vector

contains="Annotated",
representation("VIRTUAL", elementMetadata="DataTableORNULL"

Annotated

representation("VIRTUAL", metadata = "list")

2.5.5 Classes and methods



MotifDb Methods

Method	Generic Defined In	Inherited From
show	methods	
query	rtracklayer	
export	MotifDb	
subset	base	
[	base	SimpleList
[[	base	SimpleList
as.list	base	SimpleList
c	base	SimpleList
initialize	methods	.Object
length	base	SimpleList
names	base	SimpleList
names<-	base	SimpleList
elementMetadata	IRanges	Vector
elementMetadata<-	IRanges	Vector
lapply	BiocGenerics	SimpleList
mcols	IRanges	Vector
mcols<-	IRanges	Vector
sapply	BiocGenerics	List
values	IRanges	Vector

2.5.6 The query method

Of the four methods MotifDb exports, we will look at one: `query`. It expects a queryString, and a MotifDb. It returns all of the elements in the MotifList in which *any* of its metadata values match the queryString.

```
setMethod ('query', 'MotifList',
   function (object, queryString, ignore.case=TRUE) {
       indices = unique (as.integer (unlist (sapply (colnames (values (object)),
           function (colname)
              grep (queryString, values
                (object)[, colname],
                   ignore.case=ignore.case)))))
       object [indices]
       })                                                                                                  })
```

2.5.7 zzz.R

One of the idioms of R programming – the *zzz.R* file – takes advantage of the alphabetical order in which R loads and executes files. We put any code we wish to evaluate last in this file. In MotifDb, the final step of the load process initiated by `library(MotifDb)` is to load matrices and metadata from the `inst/extdata` directory, populating the `SimpleList` and `DataFrame` which lie at the heart of the MotifList. Here is the code:

```
.MotifDb = function(loadAllSources=TRUE, quiet=TRUE) {
    mdb = MotifList()
    if(loadAllSources) {
        data.path = system.file('extdata', package='MotifDb')
        data.files = dir(data.path, full.names=TRUE)
        if(length(data.files) > 0)
            for(data.file in data.files) {
                tbl.md = NA; matrices = NA;
                variables = load(data.file)
                mdb = append(mdb, MotifList(matrices, tbl.md))
                if(!quiet)
                  message(noquote(sprintf('added %s(%d) matrices, length now: %d',
                       basename(data.file), length(matrices), length(mdb))))
               } # for data.file
          if(!quiet) {
              print(table(values(mdb)$dataSource))
              }
        } # if loadAllSources
      return(mdb)
   }

.onLoad <- function(libname, pkgname) {
    MotifDb <<- .MotifDb(loadAllSources=TRUE, quiet=TRUE)
    }
```

2.5.8 Unit Tests

All columns of a normalized position frequency matrix should sum to 1.0. Each column represents one position in the target DNA to which the protein binds, with a row for each of the four possible bases: A, C, G and T. However, the column sums are not always perfect, due to experimental error, or some mistake in curation. Approximately half of the matrices obtained from UniPROBE, which are offered as pre-calculated position frequency matrices, depart slightly from perfect normalization. The test accomodates this.

```
test.allMatricesAreNormalized = function () {
```

```
matrices = MotifDb@listData
checkTrue (all (sapply (matrices, function (mtx) checkEqualsNumeric (mean (colSums (mtx)), 1.0, tolera
}
```

Chapter 3

S4 classes and methods

3.1 Introduction

The *S4 class system* is a set of facilities provided in *R* for Object Oriented (OO) programming. S4 is implemented in the *methods* package. On a fresh *R* session:

```
> sessionInfo()
```

```
...
attached base packages:
[1] stats     graphics  grDevices utils     datasets
[6] methods   base
```

R also supports an older class system, the *S3 class system*, that is much simpler and completely integrated in the language itself. *S4* is much more powerful than *S3*, but also much more complex.

3.1.1 A different OO paradigm

The OO paradigm in *S4* is different from the OO paradigm found in most other programming languages. One of the most visible differences is the syntax used to call a method on an object `x`. The *R* way (S4 and S3) is to do:

```
> foo(x, ...)
```

whereas most other programming langauges would have a syntax like `x.foo(...)`. The central concepts in *R*'s S4 system are:

Core components: *classes*[1], *generic functions*, and *methods.*
Glue: *method dispatch* (supports *simple* and *multiple* dispatch)

Those concepts are materialized in the *methods* package:

```
> ls('package:methods')
```

[1]also called *formal classes*, to distinguish them from the S3 classes or *old-style classes*

```
  [1] "@<-"                              "addNextMethod"
  [3] "allGenerics"                      "allNames"
  [5] "Arith"                            "as"
  [7] "as<-"                             "asMethodDefinition"
...
[211] "testVirtual"                      "traceOff"
[213] "traceOn"                          "tryNew"
[215] "trySilent"                        "unRematchDefinition"
[217] "validObject"                      "validSlotNames"
```

This is a rich, complex, somewhat intimidating package. The classes and methods we implement in our packages can be hard to document, especially when the class hierarchy is complicated and multiple dispatch is used.

3.1.2 S4 in *Bioconductor*

S4 is heavily used in *Bioconductor*. For example BioC 2.7 contained 1383 classes and 8397 methods defined in 200 packages (out of 419), and those numbers are growing at each new release! For the end-user: it's mostly transparent. But when something goes wrong, error messages issued by the S4 class system can be hard to understand. Also it can be hard to find the documentation for a specific method. Most *Bioconductor* packages use only a small subset of the S4 capabilities (which covers 99.99% of our needs).

3.1.3 From an end-user point of view

S4 objects can come from:

- A data set:

  ```
  > library(graph)
  > data(apopGraph)
  > apopGraph
  ```

  ```
  A graphNEL graph with directed edges
  Number of Nodes = 50
  Number of Edges = 59
  ```

- A *constructor*:

  ```
  > library(IRanges)
  > IRanges(start=c(101, 25), end=c(110, 80))
  ```

  ```
  IRanges of length 2
      start end width
  [1]   101 110    10
  [2]    25  80    56
  ```

- A coercion:

  ```
  > library(Matrix)
  > m <- matrix(3:-4, nrow=2)
  > as(m, "Matrix")
  ```

```
2 x 4 Matrix of class "dgeMatrix"
     [,1] [,2] [,3] [,4]
[1,]    3    1   -1   -3
[2,]    2    0   -2   -4
```

- A specialized high-level constructor:

```
> library(GenomicFeatures)
> makeTranscriptDbFromUCSC("sacCer2", tablename="ensGene")
```

```
TranscriptDb object:
| Db type: TranscriptDb
| Data source: UCSC
| Genome: sacCer2
| UCSC Table: ensGene
...
```

- A high-level I/O function:

```
> library(ShortRead)
> lane1 <- readFastq("path/to/my/data/", pattern="s_1_sequence.txt")
> lane1
```

```
class: ShortReadQ
length: 256 reads; width: 36 cycles
```

- Extracting part of a bigger object (typically with a *getter*):

```
> sread(lane1)
```

```
  A DNAStringSet instance of length 256
      width seq
  [1]    36 GGACTTTGTAGGATACCCTCGCTTTCCTTCTCCTGT
  [2]    36 GATTTCTTACCTATTAGTGGTTGAACAGCATCGGAC
  [3]    36 GCGGTGGTCTATAGTGTTATTAATATCAATTTGGGT
  [4]    36 GTTACCATGATGTTATTTCTTCATTTGGAGGTAAAA
  ...   ... ...
[253]    36 GTTTTACAGACACCTAAAGCTACATCGTCAACGTTA
[254]    36 GATGAACTAAGTCAACCTCAGCACTAACCTTGCGAG
[255]    36 GTTTGGTTCGCTTTGAGTCTTCTTCGGTTCCGACTA
[256]    36 GCAATCTGCCGACCACTCGCGATTCAATCATGACTT
```

How to find the right man page?

- `class?graphNEL` or equivalently ``?`graphNEL-class` `` for accessing the man page of a class
- `?qa` for accessing the man page of a generic function
- The man page for a generic might also document some or all of the methods for this generic. The *See Also:* section might give a clue. Also using `showMethods()` can be useful:

```
> showMethods("qa")
```

```
Function: qa (package ShortRead)
dirPath="character"
dirPath="list"
dirPath="ShortReadQ"
dirPath="SolexaPath"
```

- `` ?`qa,ShortReadQ-method` `` to access the man page for a particular method (might be the same man page as for the generic)
- In doubt: `??qa` will search the man pages of all the installed packages and return the list of man pages that contain the string `qa`

How to inspect objects and discover methods?

- `class()` and `showClass()`:

  ```
  > class(lane1)
  ```

  ```
  [1] "ShortReadQ"
  attr(,"package")
  [1] "ShortRead"
  ```

  ```
  > showClass("ShortReadQ")
  ```

  ```
  Class "ShortReadQ" [package "ShortRead"]

  Slots:

  Name:       quality        sread           id
  Class: QualityScore DNAStringSet   BStringSet

  Extends:
  Class "ShortRead", directly
  Class ".ShortReadBase", by class "ShortRead", distance 2

  Known Subclasses: "AlignedRead"
  ```

- `str()` for compact display of the content of an object
- `showMethods()` to discover methods
- `selectMethod()` to see the code

3.1.4 Chapter overview

Throughout this chapter you will implement a toy class, named *SNPLocations*, which is a simple container for storing SNPs in a naive and incomplete way.

Implementing an S4 class typically consists in the following steps:

1. A class definition where the name and type of each slot is specified. Unlike with other OO programming languages, the methods that will operate on this class are not part of the class definition.

2. A constructor so we can create *SNPLocations* instances. A common practice is to define an ordinary function named like the class itself for this. Note that this is not enforced by by the S4 class system, just a shared practice among the *Bioconductor* core developers. This `SNPLocations` function will take care of doing some basic argument checking and to populate the slots of the instance to be returned.
3. Some accessor methods to get values from (or set values to) the slots of a *SNPLocations* object. Note that direct slot manipulation by the end user via the `@` operator is generally not recommended. Providing our own set of accessors will hopefully discourage the user of our objects from doing this. It's also a way for us to formally specify which slots are ok to be accessed and how they should be accessed (read-only slot or read-write slot).
4. Other accessor-like methods that are not *slot* accessors (i.e. they are not getting or setting the content of a slot, strictly speaking) but are getting or setting information in the object (from or into more than 1 slot), e.g. `[` or `[<-`.
5. A `show` method, so our objects display nicely/compactly some useful information.
6. A *validity method* that will take care of checking that our *SNPLocations* objects are valid i.e. that their slots contain values that make sense individually and *as a whole*. Note that this certainly requires some extra effort whose benefits maybe aren't immediately obvious, but it is considered good practice since it makes your class implementation more robust and it pays off in the long term maintenance of your package. For this course, because of time constraints, we will implement an incomplete *validity method* for our *SNPLocations* objects.
7. Some *coercion methods* to turn our *SNPLocations* objects into other types of objects, with or without loss of information. For this course, we will implement a *coercion method* for turning a *SNPLocations* object into a data frame.
8. Other high-level methods that don't fall into any of the previous categories (i.e. not accessor, show, validity or coercion methods). Depending on the kind of object that is being implemented, those can be methods for subsetting, plotting, normalizing, generating an HTML report, etc...

This is really what we mean when we say *implementing S4 objects*.

An additional task is to integrate the class in our package, which is also an important aspect of implementing an S4 object.

So the work we need to do can be divided in 2 parts:

- Part I: Implement the *SNPLocations* class in a standalone file.
- Part II: Integrate the class implemented in Part I into a package. This is not only adding the file made in Part I to the package, it also requires modifying the `Collate` field, import the *methods* package (if not already done), modify the `NAMESPACE` file, and add a man page documenting the class. Once everything is in place, we should be able to build and check our package with `R CMD build` and `R CMD check`.

3.2 Implementing the *SNPLocations* class

3.2.1 Choosing a good design

The process of designing and implementing a new class requires that the developer spends some time thinking about:

- What s/he wants to achieve exactly with the class,
- How is the class going to be used, by who, for doing what,
- What are the typical use cases,
- What is the typical size of the data that will be manipulated, small (< 1 Mb), big (> 100 Mb), very big (> 10 Gb),
- How the class will interact with other packages and classes in CRAN/*Bioconductor*,
- How the facilities provided by the class will fit within the tools and file formats commonly used inside or outside *Bioconductor*,
- etc...

It's generally considered good design to avoid storing redundant information (although some exceptions can be made for performance considerations) and to keep things as simple as possible.

3.2.2 Class definition

For our *SNPLocations* class, we want the following slots:

- `genome`: a single string containing the name of a reference genome, e.g. `"hg19"` or `"mm10"`. It's important to make sure that the locations of our SNPs correspond to locations on that genome.
- `snpid`: a character vector of length N (where N is the number of SNPs stored in our object) containing 1 snp id per SNP.
- `chrom`: a character vector of length N containing the name of the chromosome where each SNP is located.
- `pos`: an integer vector of length N containing the position of each SNP. To keep things simple, we only want to support single-base substitutions so we don't need to store the `start` and the `end` of each SNP. Note that the almost universally adopted coordinate system on a reference genome is to report 1-based positions relative to the 5' end of the plus strand of the chromosomes.

Exercise 1
Start a new file (let's name it `SNPLocations-class.R`) and write the `setClass` statement for the SNPLocations class.

```
  setClass("SNPLocations",
      representation(
          genome="character",
          ...
          ...
      )
  )
```

Solution:

```
> setClass("SNPLocations",
+     representation(
+         genome="character",  # a single string
+         snpid="character",   # a character vector of length N
+         chrom="character",   # a character vector of length N
+         pos="integer"        # an integer vector of length N
+     )
+ )
```

3.2.3 Constructor

For the *SNPLocations* constructor, we are going to write a function that takes 4 arguments: `genome`, `snpid`, `chrom`, and `pos`. Those 4 arguments will contain the user-supplied reference genome and locations of a collection of N SNPs. Our constructor will be a simple wrapper to `new("SNPLocations", ...)`. It won't perform any checks on the user-supplied arguments (the constructor is not the best place to perform those checks, we'll see later why).

Exercise 2

a. *Add the SNPLocations constructor to the `SNPLocations-class.R` file. Note that `new("SNPLocations", ...)` must be called with named arguments. The names of the arguments must correspond to slots in the class definition. Their values must correspond to the values to assign to the slots. See `?new` for all the details.*
b. *Start R, source the `SNPLocations-class.R` file (or copy/paste its content into your session), do `showClass("SNPLocations")`, and finally, try to use the SNPLocations constructor.*

Solution:

```
> SNPLocations <- function(genome, snpid, chrom, pos)
+ {
+     new("SNPLocations", genome=genome, snpid=snpid, chrom=chrom, pos=pos)
+ }
```

Testing:

```
> mysnps <- SNPLocations("hg19",
+                        c("rs0001", "rs0002"),
+                        c("chr1", "chrX"),
+                        c(224033L, 1266886L))
> mysnps

An object of class "SNPLocations"
Slot "genome":
[1] "hg19"

Slot "snpid":
[1] "rs0001" "rs0002"

Slot "chrom":
[1] "chr1" "chrX"

Slot "pos":
[1]  224033 1266886
```

Now we are able to create *SNPLocations* objects! Keep your *R* session live for further testing on the *SNPLocations* object you just created (let's call this object `mysnps`).

Before we implement a `show` method for our *SNPLocations* objects, it's better to start by implementing a `length` method and other accessor methods so we can use them later in the `show` method (and in our code in general).

3.2.4 Implementing `length()` and other accessors

In the next exercise we will implement `length()`, `genome()`, `snpid()`, `chrom()` and `pos()` on our objects. Note that for the slot getters, we use the same name as the corresponding slot, which is a natural thing to do but is not enforced in anyway by S4.

We want to implement those accessors as *methods* for *SNPLocations* objects, not as ordinary functions. This is *the* recommended way to implement accessors. Let's distinguish between 2 situations:

- For accessors with a name that doesn't correspond to any existing function (e.g. `snpid`), we need to define a *generic* function before we can write a method for it. This is done with a `setGeneric` statement. The simplest form of the `setGeneric` statement is the following (for a generic function `foo` with a single argument):

  ```
  setGeneric("foo", function(x) standardGeneric("foo"))
  ```

- For accessors with a name that corresponds to an existing function (e.g. `length` and `genome`, defined in *base* and *GenomicRanges*, respectively), we don't need a `setGeneric` statement. (If the existing function is not already a generic function, which can be checked with `isGeneric()`, then it will be automatically turned into an *implicit* generic function.) In that case the programmer must check the signature of the existing function and make sure that s/he uses exactly the same signature in his/her method definition.

The definition of the method itself is done with a `setMethod` statement. For example, in the case of a generic function dispatching on 1 argument only (the most common situation), the `setMethod` statement looks like:

```
setMethod("foo", "SNPLocations",
    function(x)
    {
        ...
    }
)
```

Exercise 3

a. *Implement the* `length` *method for SNPLocations objects.*
b. *Load the GenomicRanges package (in order to get the* `genome()` *generic), and implement the* `genome`*,* `snpid`*,* `chrom`*, and* `pos` *accessors for SNPLocations objects.*
c. *Copy/paste the new code into your current R session and test the new methods on the* `mysnps` *objects.*

Solution:

```
> setMethod("length", "SNPLocations", function(x) length(x@snpid))
> ## The genome() generic is defined in the GenomicRanges package.
```

```
> setMethod("genome", "SNPLocations", function(x) x@genome)
> ## snpid().
> setGeneric("snpid", function(x) standardGeneric("snpid"))
> setMethod("snpid", "SNPLocations", function(x) x@snpid)
> ## chrom().
> setGeneric("chrom", function(x) standardGeneric("chrom"))
> setMethod("chrom", "SNPLocations", function(x) x@chrom)
> ## pos().
> setGeneric("pos", function(x) standardGeneric("pos"))
> setMethod("pos", "SNPLocations", function(x) x@pos)
```

Testing:

```
> length(mysnps)

[1] 2

> genome(mysnps)

[1] "hg19"

> snpid(mysnps)

[1] "rs0001" "rs0002"

> chrom(mysnps)

[1] "chr1" "chrX"

> pos(mysnps)

[1]  224033 1266886
```

3.2.5 The `show` method

`show` is a generic function defined in the *methods* package (which is also the home of the `setClass`, `setGeneric` and `setMethod` functions and the S4 class system in general). Do `?show` in your *R* session. The important bit here is that the name of the argument is `object` so that's what you need to use in your method definition.

Exercise 4

a. *Write a* `show` *method that displays something like:*

```
SNPLocations instance with 25 SNPs on genome mm9
```

Internally, use the `cat` *function to print the information, and use* `length` *to extract the number of SNPs. Also, even if you think you know the class of the object being displayed, it's better to use* `class(object)` *than to hardcode* `"SNPLocations"`*. You never know, maybe one day someone decides to extend your SNPLocations class. When this happens, your* `show` *method will work out-of-the-box on instances of the derived class (thanks to inheritance), and, because you used* `class(object)`*, it will correctly display their class.*

b. Copy/paste the definition of the `show` *method into your current R session and try to display your SNPLocations object again (by just typing the name of the object followed by <Enter>).*

Solution:

```
> setMethod("show", "SNPLocations",
+     function(object)
+         cat(class(object), "instance with", length(object),
+             "SNPs on genome", genome(object), "\n")
+ )

[1] "show"
```

Testing:

```
> mysnps

SNPLocations instance with 2 SNPs on genome hg19
```

3.2.6 The validity method

One limitation of the `setClass` statement is that the `representation` component only allows us to specify the types of the slots, but not their lengths or any other restriction that we'd want to impose.

For example, the `setClass` statement for our *SNPLocations* class just requires the `genome` slot to be a *character* vector, without imposing any restriction on its length or content. But what we really want is a single string i.e. a *character* vector of length 1 that is not an `NA`. A *SNPLocations* object with a *character* vector of length 0 or an `NA` in its `genome` slot could fairly be considered broken. Of course, we could put some sanity checks in the *SNPLocations* constructor in order to avoid this, but, a better approach is to define a *validity method* that will be in charge of those checks.

Any S4 object can be validated at any time with a call to `validObject`. By default (i.e. if no *validity method* is defined), the validation only consists in checking that the types of the slot values are *compatible* with the expected types i.e. with the types that are specified in the class definition (*compatible* here means that the slot value belongs to the specified class or to one of its subclasses). This validation is automatically performed by the low-level constructor `new` (and this is why trying to create an object with an incompatible slot value generates an error).

By defining a *validity method* for his/her objects, the developer can be much more specific about what values can go into each slot. Furthermore, it allows him/her to validate an object *as a whole* by checking that the values in the different slots are compatible with each other.

Defining a *validity method* is done with a `setValidity` statement:

```
  setValidity("SNPLocations",
      function(object)
      {
          ...
          ...
      }
  )
```

The method should return `TRUE` if the object is valid, and one or more descriptive strings if any problems are found. It should never generate an error.

In the next exercise, we implement a simple (incomplete) *validity method* for *SNPLocations* objects.

Exercise 5

a. *Implement a validity method for SNPLocations objects that will be in charge of checking that:*
    - *The* `genome` *slot is a single string (i.e. a character vector of length 1 that is not an* `NA`*);*
    - *All the other slots have the same length N (the number of SNPs).*

b. *Copy/paste the definition of the validity method into your current R session and call* `validObject` *on your SNPLocations object. Break the object by setting its* `chrom` *slot to* `"chr1"`*. Call* `validObject` *again on the object.*

c. *Is it valid to set the* `pos` *slot to* `c(25, 8)`*?*

Solution:

```
> setValidity("SNPLocations",
+     function(object)
+     {
+         if (!is.character(genome(object)) ||
+             length(genome(object)) != 1 || is.na(genome(object)))
+             return("'genome' slot must be a single string")
+         slot_lengths <- c(length(snpid(object)),
+                           length(chrom(object)),
+                           length(pos(object)))
+         if (length(unique(slot_lengths)) != 1)
+             return("lengths of slots 'snpid', 'chrom' and 'pos' differ")
+         TRUE
+     }
+ )
```

Testing:

```
> validObject(mysnps)
```

```
[1] TRUE
```

3.2.7 Coercion methods

It's often convenient for the user to be able to turn an object of a given class (the *original* class) into an object of another class (the *target* class). This transformation is called *coercion* in *R* jargon (*explicit type-casting* or *type conversion* in other programming languages). Depending on the classes that are involved, the coercion can be with or without loss of information.

When implementing an S4 class, it's good to think about potentially useful coercions that the user might need. In the case of our *SNPLocations* class for example, we'd like the user to be able to turn a *SNPLocations* object into a data frame.

R supports 2 syntaxes for performing a coercion: (1) the `as.`*targetclass*`(x)` syntax, and (2) the `as(x, "targetclass")` syntax.

The former syntax only supports a limited set of *target* classes through some predefined generic functions such as `as.logical`, `as.integer`, `as.double`, `as.numeric`, `as.complex`, `as.character`, `as.raw`, `as.vector`, `as.list`, `as.factor`, `as.matrix`, `as.array`, `as.data.frame`, etc...

The latter syntax makes use of a single generic function, the `as` generic. This is the preferred syntax when working with S4 objects: it offers greater flexibility and better integration to the S4 class system itself.

However, whenever possible, it's good to support both syntaxes to we want our user to be able to turn a *SNPLocations* object `x` into a data frame with `as.data.framex` or `as(x, "data.frame")`. For this to work, we need to implement 2 *coercion methods*: an `as.data.frame` method for *SNPLocations* objects and a `coerce` method for coercing from *SNPLocations* to *data.frame*.

The former is implemented with:

```
setMethod("as.data.frame", "SNPLocations",
    function(x, row.names=NULL, optional=FALSE, ...)
    {
        ...
    }
)
```

The latter is implemented with a `setAs` statement:

```
setAs("SNPLocations", "data.frame", function(from) as.data.frame(from))
```

The `from` argument contains the object to coerce.

Both methods must of course return the same thing, which is the coerced object. Note that the latter can be a simple wrapper to the former: there is no need to re-implement the real work done by the former in the latter.

Exercise 6

a. *Implement the 2 coercion methods from SNPLocations to data.frame.*

b. *Copy/paste the definitions of the coercion methods into your current R session and test them by doing* `as.data.frame(mysnps)` *and* `as(mysnps, "data.frame")`*.*

Solution:

```
> setMethod("as.data.frame", "SNPLocations",
+     function(x, row.names=NULL, optional=FALSE, ...)
+     {
+         ## We ignore the 'row.names' and 'optional' arguments.
+         data.frame(snpid=snpid(x), chrom=chrom(x), pos=pos(x))
+     }
+ )
```

```
[1] "as.data.frame"
```

```
> setAs("SNPLocations", "data.frame", function(from) as.data.frame(from))
```

Testing:

```
> as.data.frame(mysnps)

   snpid chrom     pos
1 rs0001  chr1  224033
2 rs0002  chrX 1266886

> as(mysnps, "data.frame")

   snpid chrom     pos
1 rs0001  chr1  224033
2 rs0002  chrX 1266886
```

3.3 Integrating the *SNPLocations* class to our package

Now we need to integrate the code produced in the previous section to our package. This is done in 4 steps.

3.3.1 Add the `SNPLocations-class.R` file to the package

Exercise 7

a. *Put the `SNPLocations-class.R` file under the `R/` folder of your package. In case you are using a revision control system like Subversion to develop your code, don't forget to add the file to the system with e.g. `svn add`.*

b. *Add the name of the new `.R` file to the `Collate` field of the `DESCRIPTION` file. The new file should be listed after any other file that contains material used in the new file and before any other file that depends on material defined in the new file.*

3.3.2 Import the required packages and modify the `NAMESPACE` file

Using S4 in a package requires that we import the *methods* package. We also need to import any other *thing* used internally in our package. For example, we need to import the `genome` generic function from the *GenomicRanges* package because we define a `genome` method in our package.

Exercise 8

a. *Make sure the methods package is in the `Imports` field of your package. For GenomicRanges, you could also put it in the `Imports` field, but, since your package is defining a `genome` method, it's a good idea to make sure that the user will also have access to the man page for the `genome` generic (located in GenomicRanges) in addition to the man page for your method (which will be located in your package). This is achived by putting GenomicRanges in the `Depends` field instead of `Imports`.*

b. *Then make the following modifications to the `NAMESPACE` file of your package:*

   - *Make sure the file contains the following directive:*

     ```
     import(methods)
     ```

     If not, add it before any other imports.

- *Import the* `genome` *generic from the GenomicRanges package:*

  ```
  importFrom(GenomicRanges,genome)
  ```
- *Export the SNPLocations class by adding the name of the class inside the* `exportClasses` *directive. Syntax:*

  ```
  exportClasses(
      Class1,
      Class2,
      ...
      ...
  )
  ```
- *In the* `export` *directive: Add the functions (non-generic and generic) defined in your package that you want to export. Note that what you export will need to be documented in a man page. The stuff that is not intended to be used directly by the user of your package should not be exported (and not documented, of course, but that doesn't mean it doesn't deserve some brief documentation in the form of a short comment in your source code).*
- *In the* `exportMethods` *directive: Add the methods you want to export (usually all the methods defined in your package, including coercion methods, but excluding validity methods). Note that the names you need to put in the directive are those of the corresponding generics with no specification of the classes for which the methods are defined. This means that if you implemented more than one method for the generic* `foo`*, then* `foo` *only needs to be listed once in the* `exportMethods` *directive:*

  ```
  exportMethods(
      ...
      foo,  # exports all the methods attached to this generic
      ...
  )
  ```

  To export the coercion methods, add `coerce` *to the* `exportMethods` *directive.*

3.3.3 Add a man page for the *SNPLocations* class

Documenting the new class and its basic functionality might not be the most exciting part of the story but, unfortunately, it's an indispensable one! To help get us motivated, let's remember that undocumented functionality is probably not going to be used, or, in the best case, will make our most adventurous users feel frustrated.

An easy approach would be to use `promptClass("SNPLocations")` which automatically generates a minimalist man page for our class. However, in our opinion, this automatic man page does not provide useful information to the user. It's also a little bit misleading since it encourages the user to create objects with direct calls to (new) (instead of using our higher-level constructor) and to manipulate slots directly (instead of using our accessors).

In our experience, using the following template for documenting our classes leads to more valuable documentation than the `promptClass` solution:

```
\name{SNPLocations-class}
\docType{class}
```

```
\alias{SNPLocations-class}
\alias{SNPLocations}
\alias{...}
\alias{...}
\alias{...}

\title{SNPLocations objects}

\description{
  ~~ A concise (1-5 lines) description of what the class is. ~~
}

\section{Constructor}{
  \describe{
    \item{}{
      \code{SNPLocations(...)}:
      ~~ A description of the constructor and its arguments. ~~
    }
  }
}

\section{Length and accessors}{
  In the code snippets below, \code{x} is a SNPLocations object.

  \describe{
    \item{}{
      \code{length(x)}:
      ~~ What the length of x is. ~~
    }
    \item{}{
      \code{accessor1(x)}:
      ~~ A description of accessor 1. ~~
    }
    \item{}{
      \code{accessor2(x)}:
      ~~ A description of accessor 2. ~~
    }

    ... etc ...

  }
}

\section{Coercion}{
  In the code snippets below, \code{x} is a SNPLocations object.
  \describe{
    \item{}{
      \code{as(x, "class1")}:
      ~~ A description of what this coercion does. ~~
    }
    \item{}{
      \code{as(x, "class2")}:
      ~~ A description of what this coercion does. ~~
    }

    ... etc ...

  }
}

\references{
  ~~ Put references to the literature/web site here. ~~
}

\author{Your Name}

\seealso{
  ~~ Put links of the form \code{\link{FUNCTIONNAME}} here ~~
  ~~ to link to other functions. ~~
  ~~ Put links of the form \code{\linkS4class{CLASSNAME}} here ~~
  ~~ to link to other classes. ~~
}

\examples{
  ~~ Put code here that illustrates at least the use of the ~~
  ~~ constructor, accessors and coercion methods (if any). ~~
}

\keyword{classes}
```

Exercise 9
Use the above template to produce the `SNPLocations-class.Rd` file. This file needs to be located under the `man/` folder of your package. In case you are using a revision control system*, don't forget to add the file to it. Note that:*

- *There must be an alias of the form*

  ```
  \alias{foo}
  ```

  for each exported function (ordinary or generic). Also there must be an alias for each exported method. The form of this alias depends on the number of arguments involved in the dispatch. It's

```
\alias{foo,Class1-method}
```

for dispatch on 1 argument (e.g. for the accessor methods), and

```
\alias{bar,Class1,Class2-method}
```

for dispatch on 2 arguments (e.g. for the coercion methods), and so on...

- *There is no alias for the validity methods (they are not exported and they don't need to be documented). What needs to be documented with great details however is what the arguments of our high-level constructor are expected to be. In the case of paths to on-disk files like for our (SNPLocations) constructor, it's also a good idea to describe what the content of those files is expected to be.*
- *There must be an alias for the* `show` *method just to avoid an* `R CMD check` *warning (see below) even though it's ok to not document the method.*
- *The* `examples` *section is probably the most important part of any man page since most users tend to go directly there without taking the time to read the whole story (either because they already know it or because they are in a hurry).*

3.3.4 Check the package

Exercise 10

a. *Run* `R CMD build` *on the package source tree. This produces a source tarball. Then run* `R CMD check` *on this source tarball and pay attention to any NOTE or WARNING that shows up. Fix them if necessary.*

b. *Install the source tarball by running* `R CMD INSTALL` *on it. Start a fresh R session, load the package, and try to use the new code. In particular, go to the new man page (*`?SNPLocations`*) so you can see what it looks like from an end-user point of view.*

If you are using a *revision control system* and are satisfied with your work so far, then it's a good time to commit it.

3.4 Extending an existing class

Like any other OO programming language, S4 lets you extend an existing class. Most of the time (but not always), the child class will have additional slots, and only those slots need to be specified in the `setClass` statement defining the child class:

```
> setClass("AnnotatedSNPs",
+     contains="SNPLocations",
+     representation(
+         geneid="character"  # a character vector of length N
+     )
+ )
```

The slots from the parent class are inherited:

```
> showClass("AnnotatedSNPs")
```

```
Class "AnnotatedSNPs" [in ".GlobalEnv"]

Slots:

Name:      geneid    genome     snpid     chrom       pos
Class: character character character character   integer

Extends: "SNPLocations"
```

The amount of work that needs to be done to bring the child class to the same level of functionality as its parent class is greatly reduced. Let's walk thru each of them.

3.4.1 Constructor

By calling the constructor for the parent class from within the constructor for the child class, we hide the implementation details of the parent class:

```
> AnnotatedSNPs <- function(genome, snpid, chrom, pos, geneid)
+ {
+     new("AnnotatedSNPs",
+         SNPLocations(genome, snpid, chrom, pos),
+         geneid=geneid)
+ }
> mysnps2 <- AnnotatedSNPs("hg19",
+                          c("rs0001", "rs0002"),
+                          c("chr1", "chrX"),
+                          c(224033L, 1266886L),
+                          c("AAU1", "SXW-23"))
```

A note about *instance* versus *object*:

```
> is(mysnps2, "AnnotatedSNPs")     # 'mysnps2' is an AnnotatedSNPs object

[1] TRUE

> is(mysnps2, "SNPLocations")      # and is also a SNPLocations object

[1] TRUE

> class(mysnps2)                   # but is *not* a SNPLocations *instance*

[1] "AnnotatedSNPs"
attr(,"package")
[1] ".GlobalEnv"
```

Exercise 11
Is `mysnps` *an AnnotatedSNPs object?*

3.4.2 `length()`, accessors, and `show` method

They all work out-of-the-box:

```
> length(mysnps2)

[1] 2

> genome(mysnps2)

[1] "hg19"

> snpid(mysnps2)

[1] "rs0001" "rs0002"

> chrom(mysnps2)

[1] "chr1" "chrX"

> pos(mysnps2)

[1]  224033 1266886

> mysnps2  # show method

AnnotatedSNPs instance with 2 SNPs on genome hg19
```

so only the `geneid()` accessor would need to be implemented:

```
> setGeneric("geneid", function(x) standardGeneric("geneid"))
> setMethod("geneid", "AnnotatedSNPs", function(x) x@geneid)
```

3.4.3 The validity method

The *validity method* for *AnnotatedSNPs* objects only needs to validate what's not already validated by the *validity method* for *SNPLocations* objects:

```
> setValidity("AnnotatedSNPs",
+     function(object) {
+         if (length(object@geneid) != length(object))
+             return("'geneid' slot must have the length of the object")
+         TRUE
+     }
+ )
```

Testing:

```
> validObject(mysnps2)  # starts by calling validity method for SNPLocations

[1] TRUE

>                        # objects internally
```

In other words, before an *AnnotatedSNPs* object can be considered valid, it must first be a valid *SNPLocations* object. This is why we sometimes say that validity methods are incremental.

3.4.4 Coercion methods

Even though, all the methods defined for *SNPLocations* objects work out-of-the-box on a *AnnotatedSNPs* object, sometimes they don't do the right thing. This is the case for example for our coercion methods to data frame:

```
> as(mysnps2, "data.frame")  # the 'geneid' slot is ignored
```

```
   snpid chrom     pos
1 rs0001  chr1  224033
2 rs0002  chrX 1266886
```

When this happens, we can *override* the current method with a more specific method:

```
> setMethod("as.data.frame", "AnnotatedSNPs",
+     function(x, row.names=NULL, optional=FALSE, ...)
+     {
+         ## Note the use of callNextMethod() to call the method for
+         ## SNPLocations objects.
+         cbind(callNextMethod(), geneid=geneid(x))
+     }
+ )
```

```
[1] "as.data.frame"
```

Testing:

```
> as.data.frame(mysnps2)
```

```
   snpid chrom     pos geneid
1 rs0001  chr1  224033   AAU1
2 rs0002  chrX 1266886 SXW-23
```

Finally, note that there is no need to implement the following method:

```
> ## NOT needed!
> #setAs("AnnotatedSNPs", "data.frame", function(from) as.data.frame(from))
```

It just works:

```
> as(mysnps2, "data.frame")
```

```
   snpid chrom     pos geneid
1 rs0001  chr1  224033   AAU1
2 rs0002  chrX 1266886 SXW-23
```

`selectMethod` can help provide us some insight on why this works:

```
> selectMethod("coerce", c(from="AnnotatedSNPs", to="data.frame"))
```

```
Method Definition:

function (from, to = "data.frame", strict = TRUE)
as.data.frame(from)

Signatures:
        from            to
target  "AnnotatedSNPs" "data.frame"
defined "SNPLocations"  "data.frame"
```

3.5 Other important S4 features

Here are a few other important S4 features not covered here:

- *Virtual* classes: equivalent to *abstract* classes in Java.
- Class unions (see `?setClassUnion`).
- Multiple inheritance: a powerful feature that should be used with caution. If used inappropriately, can lead to a class hierarchy that is hard or impossible to maintain.
- Reference classes (introduced in the next chapter).

3.6 Resources

- Man pages in the *methods* package: `?setClass`, `?showMethods`, `?selectMethod`, `?getMethod`, `?is`, `?setValidity`, `?as`.
- Note: S4 is *not* covered in the *An Introduction to R* or *The R language definition* manuals[2].
- The *Writing R Extensions* manual for details about integrating S4 classes to a package.
- The bioc-devel mailing list[3].
- The *R Programming for Bioinformatics* book by Robert Gentleman[4].

[2] http://cran.fhcrc.org/manuals.html
[3] http://bioconductor.org/help/mailing-list/
[4] http://bioconductor.org/help/publications/books/r-programming-for-bioinformatics/

Chapter 4

Reference classes

4.1 Introduction

Reference classes were introduced to *R* relatively recently. An instance of a reference class has *fields* and *methods* associated with it. The methods can reference the instance, and can modify the content of the instance. In this way reference classes seem more familiar to Java or C++ programmers. Of course *R*'s reference classes have unique features that expose some complicated issues.

A short example The following snippet illustrates a simple reference class

```
> Account <- setRefClass("Account",
+     fields=list(
+       balance = "integer"),
+     methods=list(
+       initialize = function(..., balance = 0L) {
+           callSuper(..., balance=as.integer(balance))
+       },
+       deposit = function(amount) {
+           "add amount to current balance"
+           .self$balance <- .self$balance + as.integer(amount)
+           .self
+       },
+       withdraw = function(amount) {
+           "withdraw amount from current balance, if possible"
+           if (.self$balance < amount)
+               stop("insufficient funds")
+           .self$balance <- .self$balance - as.integer(amount)
+           .self
+       },
+       show = function() {
+           cat("class:", class(.self), "\n")
+           cat("balance:", .self$balance, "\n")
+       }))
```

The example illustrates key features. Classes are created with `setRefClass`. The `setRefClass` function returns a *generator* function, which by convention we have named after the class. The class has *fields* and *methods*. The fields and methods are typically defined with the class, rather than separately (it is possible to define methods after the class has been created, but this does not seem like a good practice). Fields can contain any *R* class, including S4 and reference classes. Like S4 methods, reference classes can have `initialize` methods invoked when a new instance is created. Methods can start with a single character string to display help. Methods written on the class can reference the instance itself, as with the `.self` object in the `deposit` and `withdraw` methods. Fields and methods are accessed with `$`. Some methods are invoked as part of *R*'s normal evaluation, e.g., the `show` method for object printing.

Here is our reference class in action:

```
> acct <- Account$new(balance = 100)
> Account$help("deposit")

Call:
$deposit(amount)


add amount to current balance

> acct$balance

[1] 100

> acct$deposit(20)

class: Account
balance: 120

> acct$withdraw(100)

class: Account
balance: 20

> try(acct$withdraw(100))
> acct

class: Account
balance: 20
```

We use the generator function to instantiate an instance of the class. Direct field access is possible. Invoking a method on the instance, e.g., `deposit`, modifies the instance.

A key difference between reference classes and most other *R* data types, include S4 classes, is *reference*, rather than *copy-on-change*, semantics: note how modifying `b` does not modify `a`:

```
> a <- b <- 10
> b <- 20
> a

[1] 10
```

whereas modifying `acct1` modifies `acct2`

```
> acct1 <- acct2 <- Account$new(balance = 100)
> acct1$deposit(20)

class: Account
balance: 120

> acct2

class: Account
balance: 120
```

As experience *R* users we are probably surprised by this reference semantics, appreciating immediately the disastrous consequences such 'action at a distance' might have for an analysis and marveling that such behavior is the norm in other programming languages. Why on earth would one want to use a reference class?

Uses There are several situations where a reference class might be appropriate. One possibility is when the instance represents some objective reality, e.g., a window in a user interface, a file from which one is reading, or a 'singleton' such as package-level configuration options; it does not make sense to have copy-on-change semantics for data that must necessarily be shared by all instances referring to the same object. A second possibility is to circumvent the consequences of copy-on-change semantics and the memory inefficiencies it produces. For instance, modifying the small field in the following S4 class actually copies the entire object

```
> AA <- setClass("AA", representation(small="character", big="matrix"))
> a <- AA(small="foo", big=matrix(numeric(), 10000, 10000))
> system.time(slot(a, "small") <- "bar")  # copies 'big'

   user  system elapsed
  0.188   0.308   0.500
```

In contrast, only modified fields are copied in reference classes.

```
> B <- setRefClass("B", fields=list(small="character", big="matrix"))
> b <- B$new(small="foo", big=matrix(numeric(), 10000, 10000))
> system.time(b$small <- "bar")           # does not copy 'big'

   user  system elapsed
      0       0       0
```

This can have significant performance consequences, both in terms of speed (as shown above) and memory use.

4.2 Implementing reference classes

We have seen the basic steps required for reference class implementation. There are a number of additional salient points; a good starting points are the `?ReferenceClasses` and `?setRefClass` help pages.

4.2.1 Fields

While fields can be declared as above, through a named list of types, they may also be implemented as accessor functions; this is particularly useful when the 'field' is represented outside of *R*, e.g., as a C structure or data base reference.

```
> A <- setRefClass("A",
+     fields=list(
+       x=function(value) {
+           if (missing(value)) { ## 'get'
+               message("get")
+               1
+           } else ## 'set'
+               stop("'set' not implemented")
+       }))
> a <- A$new()
> a$x

[1] 1

> try(a$x <- 2)
```

Fields can be locked so that they are 'read only'

```
> A <- setRefClass("A", fields = list(x="numeric"))
> A$lock("x")
> a <- A$new(x=1:5)
> try(a$x <- 5:1)
```

4.2.2 Inheritance

Reference classes support inheritance

```
> A <- setRefClass("A", fields=list(a="integer"))
> B <- setRefClass("B", fields=list(b="numeric"), contains="A")
> B$new(a=1:5, b=1.41)

Reference class object of class "B"
Field "a":
[1] 1 2 3 4 5
Field "b":
[1] 1.41
```

including multiple inheritance (and inheritance from S4 and reference classes). Fields and methods can over-ride and invoke inherited methods.

```
> A <- setRefClass("A",
+     fields=list(a="integer"),
+     methods=list(value = function() { message("'A'"); .self$a}))
> B <- setRefClass("B", contains = "A",
+     methods=list(value = function() { message("'B'"); callSuper() }))
> B$new(a=1:5)$value()

[1] 1 2 3 4 5
```

All reference classes contain the class *envRefClass*. This class has several interesting methods, inheritted by all reference classes. Examples include:

`callSuper(...)` calls inheritted method.
`copy(shallow=FALSE)` create a copy of the instance.
`getRefClass()`, `getClass()` return the generator object or class definition of the class.
`show()` print the instance.
`trace(what, ...)`, `untrace(what)` enable the `trace` function on method `what`.

4.2.3 Best practices?

As a developer, it is tempting to embrace reference classes. They are conceptually easier to deal with than S4 classes, and have very appealing benefits in terms of performance. However, the reference based semantics make them exceedingly poor candidates for end users. They are appropriately deployed under a narrow set of circumstances, perhaps with some effort to leverage their benefits (e.g., efficient memory use) without exposing the underlying semantics. A recent example of this is in the *SummarizedExperiment* class of GenomicRanges, where the *Assays* slot of an S4 class is a implemented using a reference class designed to have favorable memory copying properties but not to expose reference semantics to the end user. Reference classes are also used fairly extensively in Rsamtools to represent files, which as mentioned have a natural reference semantics.

The interface to reference class fields and methods is potentially confusing to users who have been schooled in existing paradigms, especially accessors and replacement methods rather than direct access to S3 fields or S4 slots. For this reason some reference classes have been implemented behind a facade of standard *R* functions or S4 methods that dispatch to the underling class, e.g.,

```
> .A <- setRefClass("A", fields=list(value="numeric"))
> ## public interface 'A()', 'value()'
> A <- function(value = numeric(), ...)
+     .A$new(value=value, ...)
> value <- function(x, ...)
+     x$value
> a <- A(value=1:5)
> value(a)

[1] 1 2 3 4 5
```

4.2.4 Cautions?

Validity Reference classes do not support validity methods directly; one can write a validity method and invoke it manually.

```
> A <- setRefClass("A", fields=list(a="numeric"))
> xx <- setValidity("A", function(object) {
+     if (length(object$a) > 1)
+         "'a' is too long"
+     else TRUE
+ })
> A$new(a=1:5)                                # no validity checking

Reference class object of class "A"
Field "a":
[1] 1 2 3 4 5

> try(validObject(A$new(a=1:5)))
```

`initialize` The use of an `initialize` method imposes a subtle contract – derived classes may invoke `callSuper(...)`, expecting their arguments to be passed through your `initialize` method to the default `initialize` method. Further, unnamed arguments may be instances of the class itself (i.e., `new` is a copy constructor) or of an inherited class. Thus the initialize method should allow for additional arugments ..., to invoke `callSuper`, and to structure the signature so as not to capture, via matching by position, unnamed arguments:

```
> A <- setRefClass("A", fields=list(a="integer"))
> B <- setRefClass("B", fields = list(b="numeric"), contains="A",
+     methods = list(
+       initialize = function(..., b=3.14) {
+           callSuper(..., b=b)
+       }))
> B$new(a=1:5, b=1.41)

Reference class object of class "B"
Field "a":
[1] 1 2 3 4 5
Field "b":
[1] 1.41

> B$new(A$new(a=5:1), b=1.41)

Reference class object of class "B"
Field "a":
[1] 5 4 3 2 1
Field "b":
[1] 1.41
```

```
> B$new()

Reference class object of class "B"
Field "a":
integer(0)
Field "b":
[1] 3.14
```

4.3 Exercises

Exercise 12
Develop the simple bank account example above in S4 and reference classes. Reflect on the ease of development and the 'end-user' experience.

Chapter 5

Accessing Data: Data Base and Web Resources

5.1 Introduction

The most common interface for retrieving data in *Bioconductor* is now the `select` method. The interface provides a simple way of extracting data.

There are really 4 methods that work together to allow a `select` interface. The 1st one is `cols`, which tells you about what kinds of values you can retrieve as columns in the final result.

```
> library(Homo.sapiens)
> cols(Homo.sapiens)
```

```
 [1] "GOID"         "TERM"         "ONTOLOGY"     "DEFINITION"   "ENTREZID"
 [6] "PFAM"         "IPI"          "PROSITE"      "ACCNUM"       "ALIAS"
[11] "CHR"          "CHRLOC"       "CHRLOCEND"    "ENZYME"       "MAP"
[16] "PATH"         "PMID"         "REFSEQ"       "SYMBOL"       "UNIGENE"
[21] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"     "UNIPROT"
[26] "GO"           "EVIDENCE"     "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"
[31] "OMIM"         "UCSCKG"       "CDSID"        "CDSNAME"      "CDSCHROM"
[36] "CDSSTRAND"    "CDSSTART"     "CDSEND"       "EXONID"       "EXONNAME"
[41] "EXONCHROM"    "EXONSTRAND"   "EXONSTART"    "EXONEND"      "GENEID"
[46] "TXID"         "EXONRANK"     "TXNAME"       "TXCHROM"      "TXSTRAND"
[51] "TXSTART"      "TXEND"
```

The next method is `keytypes` which tells you the kinds of things that can be used as keys.

```
> keytypes(Homo.sapiens)
```

```
 [1] "GOID"         "ENTREZID"     "PFAM"         "IPI"          "PROSITE"
 [6] "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"       "CHRLOCEND"
[11] "ENZYME"       "MAP"          "PATH"         "PMID"         "REFSEQ"
```

```
[16] "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS"
[21] "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"
[26] "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "OMIM"         "UCSCKG"
[31] "GENEID"       "TXID"         "TXNAME"       "EXONID"       "EXONNAME"
[36] "CDSID"        "CDSNAME"
```

The third method is `keys` which is used to retrieve all the viable keys of a particular type.

```
> k <- head(keys(Homo.sapiens,keytype="ENTREZID"))
> k
```

```
[1] "1"  "2"  "3"  "9"  "10" "11"
```

And finally there is `select`, which extracts data by using values supplied by the other method.

```
> result <- select(Homo.sapiens, keys=k,
+                  cols=c("TXNAME","TXSTART","TXSTRAND"),
+                  keytype="ENTREZID")
> head(result)
```

```
  ENTREZID     TXNAME TXSTRAND  TXSTART
1        1 uc002qsd.4        - 58858172
2        1 uc002qsf.2        - 58859832
3        2 uc001qvk.1        -  9220304
4        2 uc009zgk.1        -  9220304
5        3 uc021qum.1        -  9381129
6        9 uc010ltd.3        + 18027971
```

But why would we want to implement these specific methods? It's a fair question. Why would we want to write a select interface for our annotation data? Why not just save a .rda file to the data directory and be done with it? There are basically two reasons for this. The 1st reason is convenience for end users. When your end users can access your data using the same four methods that they use everywhere else, they will have a more effortless time retrieving their data. And things that benefit your users benefit you.

The second reason is that by enabling a consistent interface across all annotation resources, we allow for things to be used in a programmatic manner. By implementing a select interface, we are creating a universal API for the whole project.

Lets look again at the example I described above and think about what is happening. The *Homo.sapiens* package is able to integrate data from many different resources largely because the separate resources all implemented a select method. This allows the *OrganismDbi* package to pull together resources from *org.Hs.eg.db*, *GO.db* and *TxDb.Hsapiens.UCSC.hg19.knownGene*.

If these packages all exposed different interfaces for retrieving the data, then it would be a lot more challenging to retrieve it, and writing general code that retrieved the appropriate data would be a lost cause. So implementing a set of select methods is a way to convert your package from a data file into an actual resource.

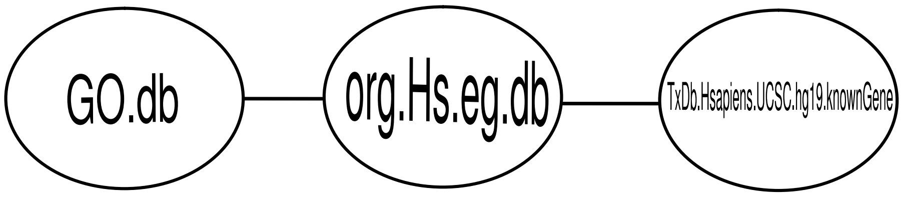


Figure 5.1: Packages and relationships represented by the Homo.sapiens package

5.2 Creating other kinds of Annotation packages

A few more automated options already exist for generating specific kinds of annotation packages. For users who seek to make custom chip packages, users should see the *SQLForge: An easy way to create a new annotation package with a standard database schema.* in the *AnnotationForge* package. And, for users who seek to make a probe package, there is another vignette called *Creating probe packages* that is also in the *AnnotationForge* package. And finally, for custom organism packages users should look at the manual page for `makeOrgPackageFromNCBI`. This function will attempt to make you an simplified organism package from NCBI resources. However, this function is not meant as a way to refresh annotation packages between releases. It is only meant for people who are working on less popular model organisms (so that annotations can be made available in this format).

But what if you had another kind of web resource or database and you wanted to expose it to the world using something like this new `select` method interface? How could you go about this?

5.3 Retrieving data from a web resource

If you choose to expose a web resource, then you will need to learn some skills for retrieving that data from the web. The *R* programming language has tools that can help you interact with web resources, pulling down files that are tab-delimited or formatted as XML etc. *R* pacakges such as *XML* and *RJSONIO* can help parse what you retrieve. In this section we retrieve data in both tab-delimited and XML format from the Uniprot web service and demonstrate how you can expose resources like this for your own purposes.

These days many web services are exposed using a representational state transfer or RESTful interface. An example of this are the services offered at Uniprot. Starting with the Uniprot base URI you can add details to simply indicate what it is that you wish to retrieve.

So in the case of Uniprot the base URI for the service we want today is this:

```
http://www.uniprot.org/uniprot/
```

This URI can be extended to retrieve individual records by specifying a query argument like this:

```
http://www.uniprot.org/uniprot/?query=P13368
```

We can then request multiple records like this:

```
http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4
```

And we can ask that the records be returned to us in tabular form by adding another argument like this.

```
http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4&format=tab
```

As you might guess, each RESTful interface is a little different, but you can easily see how once you read the documentation for a given RESTful interface, you can start to retrieve the data in *R*. Here is an example.

```
>    uri <- 'http://www.uniprot.org/uniprot/?query='
>    ids <- c('P13368', 'Q6GZX4')
>    idStr <- paste(ids, collapse="+or+")
>    format <- '&format=tab'
>    fullUri <- paste0(uri,idStr,format)
>    read.delim(fullUri)
```

```
   Entry  Entry.name   Status                        Protein.names
1 Q6GZX4  001R_FRG3G reviewed Putative transcription factor 001R
2 P13368 7LESS_DROME reviewed    Protein sevenless (EC 2.7.10.1)
          Gene.names                            Organism Length
1           FV3-001R Frog virus 3 (isolate Goorha) (FV-3)    256
2 sev HD-265 CG18085  Drosophila melanogaster (Fruit fly)   2554
```

Exercise 13
If you use the columns argument you can also specify which columns you want returned. So for example, you can choose to only have the sequence and id columns returned like this:

```
http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4&format=tab&columns=id,sequence
```

Use this detail about the Uniprot web service along with what was learned above to write a function that takes a character vector of uniprot IDs and another character vector of columns arguments and then returns the appropriate values. Be careful to filter out any extra records that the service returns.

Solution:

```
> getUniprotGoodies  <- function(query, cols)
+ {
+     ## query and cols start as a character vectors
+     qstring <- paste(query, collapse="+or+")
+     cstring <- paste(cols, collapse=",")
+     uri <- 'http://www.uniprot.org/uniprot/?query='
+     fullUri <- paste0(uri,qstring,'&format=tab&columns=',cstring)
+     dat <- read.delim(fullUri, stringsAsFactors=FALSE)
+     ## now remove things that were not in the specific original query...
+     dat <- dat[dat[,1] %in% query,]
+     dat
+ }
```

5.3.1 Parsing XML

Data for the previous example were downloaded from Uniprot in tab-delimited format. This is a convienent output to work with but unfortunately not always available. XML is still very common and it is useful to have some familiarity with parsing it. In this section we give a brief overview to using the *XML* package for navigating XML data.

The *XML* package provides functions to parse XML in both the tree-based DOM (document object model) or the event-driven SAX (Simple API for XML). We will use the DOM approach. The XML is first parsed into a tree-structure where the different elements of the data are nodes. The elements are processed by traversing the tree and generating a user-level representation of the nodes. XPath syntax is used to traverse the nodes. A detailed description of XPath can be found at `http://www.w3.org/xml`.

Retrieve the data: Data will be retrieved for the same id's as in the previous example. Unlike tab-delimited, the XML queries cannot be subset by column so the full record will be returned for each id. Details for what is possible with each type of data retrieval are found at `http://www.uniprot.org/faq/28`.

Parse the XML into a tree structure with `xmlTreeParse`. When `useInternalNodes=TRUE` and no `handlers` are specified the return value is a reference to C-level nodes. This storage mode allows us to traverse the tree of data in C instead of R objects.

```
> library(XML)
> uri <- "http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4&format=xml"
> xml <- xmlTreeParse(uri, useInternalNodes=TRUE)
```

XML namespace: XML pages can have namespaces which facilatate the use of different XML vocabularies by resolving conflicts arrising from identical tags. Namepaces are represented by a uri pointing to an XML schema page. When a namespace is defined on a node in an XML document it must be included in the XPath expression.

Use the `xmlNamespaceDefinitions` function to check if the XML has a namespace.

```
> defs <- xmlNamespaceDefinitions(xml, recurisve=TRUE)
> defs

[[1]]
$id
[1] ""

$uri
[1] "http://uniprot.org/uniprot"

$local
[1] TRUE

attr(,"class")
[1] "XMLNamespaceDefinition"

$xsi
$id
[1] "xsi"

$uri
[1] "http://www.w3.org/2001/XMLSchema-instance"

$local
[1] TRUE

attr(,"class")
[1] "XMLNamespaceDefinition"

attr(,"class")
[1] "XMLNamespaceDefinitions"
```

The uri's present in this listing confirm there is a namespace. Alternatively we could have looked at the XML nodes for declarstions of the form `xmlns:myNamespace="http://www.namspace.org"`. We organize the namespaces and will use them directly in parsing.

```
> ns <- structure(sapply(defs, function(x) x$uri), names=names(defs))
```

Parsing with XPath: There are two high level 'entry' nodes which represent the two id's requested in the original query.

```
> entry <- getNodeSet(xml, "//ns:entry", "ns")
> xmlSize(entry)

[1] 2
```

List the attributes of the top nodes and extract the names.

```
> nms <- xpathSApply(xml, "//ns:entry/ns:name", xmlValue, namespaces="ns")
> attrs <- xpathApply(xml, "//ns:entry", xmlAttrs, namespaces="ns")
> names(attrs) <- nms
> attrs

$`001R_FRG3G`
     dataset      created     modified      version
"Swiss-Prot" "2011-06-28" "2012-04-18"         "24"

$`7LESS_DROME`
     dataset      created     modified      version
"Swiss-Prot" "1990-01-01" "2012-10-03"        "134"
```

Inspect the direct children of each node.

```
> fun1 <- function(elt) unique(names(xmlChildren(elt)))
> xpathApply(xml, "//ns:entry", fun1, namespaces="ns")

[[1]]
 [1] "accession"         "name"              "protein"           "gene"
 [5] "organism"          "organismHost"      "reference"         "comment"
 [9] "dbReference"       "proteinExistence"  "keyword"           "feature"
[13] "sequence"

[[2]]
 [1] "accession"         "name"              "protein"           "gene"
 [5] "organism"          "reference"         "comment"           "dbReference"
 [9] "proteinExistence"  "keyword"           "feature"           "evidence"
[13] "sequence"
```

Query Q6GZX4 has 2 'feature' nodes and query P13368 has 48.

```
> Q6GZX4 <- "//ns:entry[ns:accession='Q6GZX4']/ns:feature"
> xmlSize(getNodeSet(xml, Q6GZX4, namespaces="ns"))

[1] 2

> P13368 <- "//ns:entry[ns:accession='P13368']/ns:feature"
> xmlSize(getNodeSet(xml, P13368, namespaces="ns"))

[1] 48
```

List all possible values for the 'type' attribute of the 'feature' nodes.

```
> path <- "//ns:feature"
> unique(xpathSApply(xml, path, xmlGetAttr, "type", namespaces="ns"))
```

```
 [1] "chain"                                "compositionally biased region"
 [3] "topological domain"                   "transmembrane region"
 [5] "domain"                               "repeat"
 [7] "nucleotide phosphate-binding region" "active site"
 [9] "binding site"                         "modified residue"
[11] "glycosylation site"                   "mutagenesis site"
[13] "sequence conflict"
```

For query P13368 extract the features with 'type=sequence conflict'.

```
> path <- "//ns:entry[ns:accession='P13368']/ns:feature[@type='sequence conflict']"
> data.frame(t(xpathSApply(xml, path, xmlAttrs, namespaces="ns")))
```

```
               type                    description ref
1 sequence conflict          In Ref. 1; AAA28882.   1
2 sequence conflict          In Ref. 3; AAF47992.   3
3 sequence conflict          In Ref. 3; AAF47992.   3
4 sequence conflict          In Ref. 3; AAF47992.   3
5 sequence conflict          In Ref. 3; AAF47992.   3
6 sequence conflict In Ref. 2; CAA31960/CAB55310.   2
7 sequence conflict          In Ref. 1; AAA28882.   1
```

Put the sequence information in an AAStringSet and add the names we extracted previously.

```
> library(Biostrings)
> path <- "//ns:entry/ns:sequence"
> seqs <- xpathSApply(xml, path, xmlValue, namespaces="ns")
> aa <- AAStringSet(unlist(lapply(seqs, function(elt) gsub("\n", "", elt)),
+     use.names=FALSE))
> names(aa) <- nms
> aa
```

```
  A AAStringSet instance of length 2
    width seq                                               names
[1]   256 MAFSAEDVLKEYDRRRRMEALLL...KGVLYDDSFRKIYTDLGWKFTPL 001R_FRG3G
[2]  2554 MTMFWQQNVDHQSDEQDKQAKGA...NLTLREVPLKDKQLYANEGVSRL 7LESS_DROME
```

5.4 Setting up a package to expose a web service

In order to expose a web service using select, you will need to create an object that will be loaded at the time when the package is loaded. Unlike with a database, the purpose of this object is pretty much purely for dispatch. We just need `select` and it's friends to know which select method to call

The first step is to create an object. Creating an object is simple enough:

```
> setClass("uniprot", representation(name="character"),
+          prototype(name="uniprot"))
```

Once you have a class defined, all you need is to make an instance of this class. Making an instance is easy enough:

```
>     uniprot <- new("uniprot")
```

But of course it's a little more complicated because one of these objects will need to be spawned up whenever our package loads. This is acclomplished by calling the `.onLoad` function in the zzz.R file. The following code will create an object, and then assign it to the package namespace as the package loads.

```
> .onLoad <- function(libname, pkgname)
+ {
+     ns <- asNamespace(pkgname)
+     uniprot <- new("uniprot")
+     assign("uniprot", uniprot, envir=ns)
+     namespaceExport(ns, "uniprot")
+ }
```

5.5 Creating package accessors for a web service

At this point you have all that you need to know in order to implement `keytype`,`cols`,`keys` and `select` for your package. In this section we will explore how you could implement some of these if you were making a package that exposed uniprot.

5.5.1 Example: creating `keytypes` and `cols` methods

The `keytype` and `cols` methods are always the 1st ones you should implement. They are the easiest, and their existence is required to be able to use `keys` or `select`. In this simple case we only have one value that can be used as a keytype, and that is a UNIPROT ID.

```
> setMethod("keytypes", "uniprot",function(x){return("UNIPROT")})
```

```
[1] "keytypes"
```

```
> uniprot <- new("uniprot")
> keytypes(uniprot)
```

```
[1] "UNIPROT"
```

So what about cols? Well it's not a whole lot more complicated in this case since we are limited to things that we can return from the web service. Since this is just an example, lets limit it to the following fields: "ID", "SEQUENCE", "ORGANISM".

```
> setMethod("cols", "uniprot",
+           function(x){return(c("ID", "SEQUENCE", "ORGANISM"))})
```

```
[1] "cols"

> cols(uniprot)

[1] "ID"       "SEQUENCE" "ORGANISM"
```

Also, notice how for both `keytypes` and `cols` I am using all capital letters. This is a style adopted throughout the project.

5.5.2 Example 2: creating a `select` method

At this point we have enough to be able to make a select method.

Exercise 14
Using what you have learned above, and the helper function from earlier, define a select method. This select method will have a default `keytype` *of "UNIPROT".*

Solution:

```
> .select <- function(x, keys, cols){
+     colsTranslate <- c(id='ID', sequence='SEQUENCE', organism='ORGANISM')
+     cols <- names(colsTranslate)[colsTranslate %in% cols]
+     getUniprotGoodies(query=keys, cols=cols)
+ }
> setMethod("select", "uniprot",
+     function(x, keys, cols, keytype)
+ {
+     .select(keys=keys, cols=cols)
+ })

[1] "select"

> select(uniprot, keys=c("P13368","P20806"), cols=c("ID","ORGANISM"))

   Entry                            Organism
1 P13368 Drosophila melanogaster (Fruit fly)
2 P20806      Drosophila virilis (Fruit fly)
```

5.6 Retrieving data from a database resource

If your package is retrieving data from a database, then there are some additional skills you will need to be able to interface with this database from *R*. This section will introduce you to those skills.

5.6.1 Getting a connection

If all you know is the name of the SQLite database, then to get a DB connection you need to do something like this:

```
> drv <- SQLite()
> library("org.Hs.eg.db")
> con <- dbConnect(drv, dbname=system.file("extdata", "org.Hs.eg.sqlite",
+                         package = "org.Hs.eg.db"))
> con
> dbDisconnect(con)
```

But in our case the connection has already been created here as part of the object that was generated when the package was loaded:

```
> require(hom.Hs.inp.db)
> str(hom.Hs.inp.db)
```

```
Reference class 'InparanoidDb' [package "AnnotationDbi"] with 2 fields
 $ conn       :Formal class 'SQLiteConnection' [package "RSQLite"] with 1 slots
  .. ..@ Id:<externalptr>
 $ packageName: chr "hom.Hs.inp.db"
 and 11 methods,
```

So we can do something like below:

```
> hom.Hs.inp.db$conn
```

```
<SQLiteConnection: DBI CON (3250, 11)>
```

```
> ## or better we can use a helper function to wrap this:
> AnnotationDbi:::dbConn(hom.Hs.inp.db)
```

```
<SQLiteConnection: DBI CON (3250, 11)>
```

```
> ## or we can just call the provided convenience function
> ## from when this package loads:
> hom.Hs.inp_dbconn()
```

```
<SQLiteConnection: DBI CON (3250, 9)>
```

5.6.2 Getting data out

Now we just need to get our data out of the DB. There are several useful functions for doing this. Most of these come from the *RSQLite* or *DBI* packages. For the sake of simplicity, I will only discuss those that are immediately useful for exploring and extracting data from a database in this vignette. One pair of useful methods are the `dbListTables` and `dbListFields` which are useful for exploring the schema of a database.

```
> con <- AnnotationDbi:::dbConn(hom.Hs.inp.db)
> head(dbListTables(con))
```

```
[1] "Acyrthosiphon_pisum"   "Aedes_aegypti"         "Anopheles_gambiae"
[4] "Apis_mellifera"        "Arabidopsis_thaliana"  "Aspergillus_fumigatus"
```

```
> dbListFields(con, "Mus_musculus")
```

```
[1] "inp_id"      "clust_id"    "species"     "score"       "seed_status"
```

For actually executing SQL to retrieve data, you probably want to use something like `dbGetQuery`. The only caveat is that this will actually require you to know a little SQL.

```
> dbGetQuery(con, "SELECT * FROM metadata")
```

```
             name                                                   value
1   INPSOURCEDATE                                             29-Apr-2009
2   INPSOURCENAME                                      Inparanoid Orthologs
3    INPSOURCEURL http://inparanoid.sbc.su.se/download/current/sqltables/
4        DBSCHEMA                                           INPARANOID_DB
5        ORGANISM                                            Homo sapiens
6         SPECIES                                                   Human
7         package                                          AnnotationDbi
8         Db type                                           InparanoidDb
9 DBSCHEMAVERSION                                                     2.1
```

5.6.3 Some basic SQL

The good news is that SQL is pretty easy to learn. Especially if you are primarily interested in just retrieving data from an existing database. Here is a quick run-down to get you started on writing simple SELECT statements. Consider a table that looks like this:

Table sna

foo	bar
1	baz
2	boo

This statement:

```
    SELECT bar FROM sna;
```

Tells SQL to get the ”bar” field from the ”foo” table. If we wanted the other field called ”sna” in addition to ”bar”, we could have written it like this:

```
    SELECT foo, bar FROM sna;
```

Or even this (* is a wildcard character here)

```
    SELECT * FROM sna;
```

Now lets suppose that we wanted to filter the results. We could also have said something like this:

```
    SELECT * FROM sna WHERE bar='boo';
```

That query will only retrieve records from foo that match the criteria for bar. But there are two other things to notice. First notice that a single = was used for testing equality. Second notice that I used single quotes to demarcate the string. I could have also used double quotes, but when working in *R* this will prove to be less convenient as the whole SQL statement itself will frequently have to be wrapped as a string.

What if we wanted to be more general? Then you can use LIKE. Like this:

```
SELECT * FROM sna WHERE bar LIKE 'boo\%';
```

That query will only return records where bar starts with ”boo”, (the % character is acting as another kind of wildcard in this context).

You will often find that you need to get things from two or more different tables at once. Or, you may even find that you need to combine the results from two different queries. Sometimes these two queries may even come from the same table. In any of these cases, you want to do a join. The simplest and most common kind of join is an inner join. Lets suppose that we have two tables:

Table sna

foo	bar
1	baz
2	boo

Table fu

foo	bo
1	hi
2	ca

And we want to join them where the records match in their corresponding ”foo” columns. We can do this query to join them:

```
SELECT * FROM sna,fu WHERE sna.foo=fu.foo;
```

Something else we can do is tidy this up by using aliases like so:

```
SELECT * FROM sna AS s,fu AS f WHERE s.foo=f.foo;
```

This last trick is not very useful in this particular example since the query ended up being longer than we started with, but is still great for other cases where queries can become really long.

5.6.4 Exploring the SQLite database from *R*

Now that we know both some SQL and also about some of the methods in *DBI* and *RSQLite* we can begin to explore the underlying database from *R*. How should we go about this? Well the 1st thing we always want to know are what tables are present. We already know how to learn this:

```
> head(dbListTables(con))

[1] "Acyrthosiphon_pisum"   "Aedes_aegypti"         "Anopheles_gambiae"
[4] "Apis_mellifera"        "Arabidopsis_thaliana"  "Aspergillus_fumigatus"
```

And we also know that once we have a table we are curious about, we can then look up it’s fields using `dbListFields`

```
> dbListFields(con, "Apis_mellifera")

[1] "inp_id"      "clust_id"    "species"     "score"       "seed_status"
```

And once we know something about which fields are present in a table, we can compose a SQL query. perhaps the most straightforward query is just to get all the results from a given table. We know that the SQL for that should look like:

```
    SELECT * FROM Apis_mellifera;
```

So we can now call a query like that from R by using `dbGetQuery`:

```
> head(dbGetQuery(con, "SELECT * FROM Apis_mellifera"))
```

```
           inp_id clust_id species score seed_status
1     XP_623957.2        1   APIME     1        100%
2 ENSP00000262442        1   HOMSA     1         99%
3 ENSP00000300671        1   HOMSA 0.095            
4  XP_001121322.1        2   APIME     1        100%
5 ENSP00000265104        2   HOMSA     1        100%
6 ENSP00000333363        2   HOMSA 0.236            
```

Exercise 15
Now use what you have learned to explore the hom.Hs.inp.db database. The formal scientific name for one of the mosquitoes that carry the malaria parasite is Anopheles gambiae. Now find the table for that organism in the hom.Hs.inp.db database and extract it into R. How many species are present in this table? Inparanoid uses a five letter designation for each species that is composed of the 1st 2 letters of the genus followed by the 1st 3 letters of the species. Using this fact, write a SQL query that will retrieve only records from this table that are from humans (Homo sapiens).

Solution:

```
> head(dbGetQuery(con, "SELECT * FROM Anopheles_gambiae"))
> ## Then only retrieve human records
> ## Query: SELECT * FROM Anopheles_gambiae WHERE species='HOMSA'
> head(dbGetQuery(con, "SELECT * FROM Anopheles_gambiae WHERE species='HOMSA'"))
> dbDisconnect(con)
```

5.7 Setting up a package to expose a SQLite database object

For the sake of simplicity, lets look at an existing example of this in the hom.Hs.inp.db package. If you download this tarball from the website you can see that it contains a .sqlite database inside of the inst/extdata directory. There are a couple of important details though about this database. The 1st is that we recommend that the database have the same name as the package, but end with the extension .sqlite. The second detail is that we recommend that the metadata table contain some important fields. This is the metadata from the current hom.Hs.inp.db package.

```
             name                                              value
1   INPSOURCEDATE                                        29-Apr-2009
2   INPSOURCENAME                                Inparanoid Orthologs
```

```
3    INPSOURCEURL http://inparanoid.sbc.su.se/download/current/sqltables/
4        DBSCHEMA                                            INPARANOID_DB
5        ORGANISM                                             Homo sapiens
6         SPECIES                                                    Human
7         package                                            AnnotationDbi
8         Db type                                             InparanoidDb
9 DBSCHEMAVERSION                                                      2.1
```

As you can see there are a number of very useful fields stored in the metadata table and if you list the equivalent table for other packages you will find even more useful information than you find here. But the most important fields here are actually the ones called ”package” and ”Db type”. Those fields specify both the name of the package with the expected class definition, and also the name of the object that this database is expected to be represented by in the R session respectively. If you fail to include this information in your metadata table, then `loadDb` will not know what to do with the database when it is called. In this case, the class definition has been stored in the *AnnotationDbi* package, but it could live anywhere you need it too. By specifying the metadata field, you enable `loadDb` to find it.

Once you have set up the metadata you will need to create a class for your package that extends the *AnnotationDb* class. In the case of the hom.Hs.inp.db package, the class is defined to be a *InparanoidDb* class. This code is inside of *AnnotationDbi*.

```
> .InparanoidDb <-
+     setRefClass("InparanoidDb", contains="AnnotationDb")
```

Finally the `.onLoad` call for your package will have to contain code that will call the `loadDb` method. This is what it currently looks like in the *hom.Hs.inp.db* package.

```
> sPkgname <- sub(".db$","",pkgname)
> db <- loadDb(system.file("extdata", paste(sPkgname,
+                ".sqlite",sep=""), package=pkgname, lib.loc=libname),
+                packageName=pkgname)
> dbNewname <- AnnotationDbi:::dbObjectName(pkgname,"InparanoidDb")
> ns <- asNamespace(pkgname)
> assign(dbNewname, db, envir=ns)
> namespaceExport(ns, dbNewname)
```

When the code above is run (at load time) the name of the package (AKA ”pkgname”, which is a parameter that will be passed into `.onLoad`) is then used to derive the name for the object. Then that name, is used by `onload` to create an *InparanoidDb* object. This object is then assigned to the namespace for this package so that it will be loaded for the user.

5.8 Creating package accessors for databases

At this point, all that remains is to create the means for accessing the data in the database. This should prove a lot less difficult than it may initially sound. For the new interface, only the four methods that were described earlier are really required: `cols`,`keytypes`,`keys` and `select`.

In order to do this you need to know a small amount of SQL and a few tricks for accessing the database from R. The point of providing these 4 accessors is to give users of these packages a more unified experience when retrieving data from the database. But other kinds of accessors (such as those provided for the *TranscriptDb* objects) may also be warranted.

5.8.1 Examples: creating a `cols` and `keytypes` method

Now lets suppose that we want to define a `cols` method for our *hom.Hs.inp.db* object. And lets also suppose that we want is for it to tell us about the actual organisms for which we can extract identifiers. How could we do that?

```
> .cols <- function(x)
+ {
+     con <- AnnotationDbi:::dbConn(x)
+     list <- dbListTables(con)
+     ## drop unwanted tables
+     unwanted <- c("map_counts","map_metadata","metadata")
+     list <- list[!list %in% unwanted]
+     ## Then just to format things in the usual way
+     list <- toupper(list)
+     dbDisconnect(con)
+     list
+ }
> ## Then make this into a method
> setMethod("cols", "InparanoidDb", .cols(x))
> ## Then we can call it
> cols(hom.Hs.inp.db)
```

Notice again how I formatted the output to all uppercase characters? This is just done to make the interface look consistent with what has been done before for the other `select` interfaces. But doing this means that we will have to do a tiny bit of extra work when we implement out other methods.

Exercise 16
Now use what you have learned to try and define a method for `keytypes` *on hom.Hs.inp.db. The keytypes method should return the same results as cols (in this case). What if you needed to translate back to the lowercase table names? Also write an quick helper function to do that.*

Solution:

```
> setMethod("keytypes", "InparanoidDb", .cols(x))
> ## Then we can call it
> keytypes(hom.Hs.inp.db)
> ## refactor of .cols
> .getLCcolnames <- function(x)
+ {
+     con <- AnnotationDbi:::dbConn(x)
+     list <- dbListTables(con)
+     ## drop unwanted tables
+     unwanted <- c("map_counts","map_metadata","metadata")
+     list <- list[!list %in% unwanted]
+     dbDisconnect(con)
+     list
+ }
> .cols <- function(x)
+ {
+     list <- .getLCcolnames(x)
+     ## Then just to format things in the usual way
+     toupper(list)
+ }
> ## Test:
> cols(hom.Hs.inp.db)
> ## new helper function:
> .getTableNames <- function(x)
+ {
+     LC <- .getLCcolnames(x)
+     UC <- .cols(x)
+     names(UC) <- LC
+     UC
+ }
> .getTableNames(hom.Hs.inp.db)
```

5.8.2 Example: creating a `keys` method

Exercise 17
Now define a method for `keys` *on hom.Hs.inp.db. The keys method should return the keys from a given organism based on the appropriate keytype. Since each table has rows that correspond to both human and non-human IDs, it will be necessary to filter out the human rows from the result*

Solution:

```
> .keys <- function(x, keytype)
+ {
+     ## translate keytype back to table name
+     tabNames <- .getTableNames(x)
+     lckeytype <- names(tabNames[tabNames %in% keytype])
+     ## get a connection
+     con <- AnnotationDbi:::dbConn(x)
+     sql <- paste("SELECT inp_id FROM",lckeytype, "WHERE species!='HOMSA'")
+     res <- dbGetQuery(con, sql)
+     res <- as.vector(t(res))
+     dbDisconnect(con)
+     res
+ }
> setMethod("keys", "InparanoidDb", .keys(x, keytype))
> ## Then we can call it
> keys(hom.Hs.inp.db, "TRICHOPLAX_ADHAERENS")
```

5.9 Creating a database resource from available data

Sometimes you may have a lot of data that you want to organize into a database. Or you may have another existing database that you wish to convert into a SQLite database. This section will deal with some simple things you can do to create and import a SQLite database of your own.

5.9.1 Making a new connection

First, lets close the connection to our other DB:

```
>    dbDisconnect(con)
```

```
[1] TRUE
```

Then lets make a new database. Notice that we specify the database name with ”dbname” This allows it to be written to disc instead of just memory.

```
> drv <- dbDriver("SQLite")
> dbname <- file.path(tempdir(), "myNewDb.sqlite")
> con <- dbConnect(drv, dbname=dbname)
```

5.9.2 Importing data

Imagine that we want to reate a database and then put a table in it called genePheno to store the genes mutated and a phenotypes associated with each. Plan for genePheno to hold the following gene IDs and phenotypes (as a toy example):

```
> data = data.frame(id=c(1,2,9),
+                   string=c("Blue",
+                            "Red",
+                            "Green"),
+                   stringsAsFactors=FALSE)
```

Making the table is very simple, and just involves a create table statement.

```
 CREATE Table genePheno (id INTEGER, string TEXT);
```

The SQL create table statement just indicates what the table is to be called, as well as the different fields that will be present and the type of data each field is expected to contain.

```
> dbGetQuery(con, "CREATE Table genePheno (id INTEGER, string TEXT)")
```

```
NULL
```

But putting the data into the database is a little bit more delicate. We want to take control over which columns we want to insert from our `data.frame`. Fortunately, the RSQLite package provides these facilities for us.

```
> names(data) <- c("id","string")
> sql <- "INSERT INTO genePheno VALUES ($id, $string)"
> dbBeginTransaction(con)

[1] TRUE

> dbGetPreparedQuery(con, sql, bind.data = data)

NULL

> dbCommit(con)

[1] TRUE
```

Please notice that we want to use strings instead of factors in our data.frame. If you insert the data as factors, you may not be happy with what ends up in the DB.

5.9.3 Attaching other database resources

In SQLite it is possible to attach another database to your session and then query across both resources as if they were the same DB.

The SQL what we want looks quite simple:

```
ATTACH "TxDb.Hsapiens.UCSC.hg19.knownGene.sqlite" AS db;
```

So in R we need to do something similar to this:

```
> db <- system.file("extdata", "TxDb.Hsapiens.UCSC.hg19.knownGene.sqlite",
+                    package="TxDb.Hsapiens.UCSC.hg19.knownGene")
> dbGetQuery(con, sprintf("ATTACH '%s' AS db",db))

NULL
```

Here we have attached a DB from one of the packages that this vignette required you to have installed, but we could have attached any SQLite database that we provided a path to.

Once we have attached the database, we can join to it's tables as if they were in our own database. All that is required is a prefix, and some knowledge about how to do joins in SQL. In the end the SQL to take advantage of the attached database looks like this:

```
SELECT * FROM db.gene AS dbg, genePheno AS gp
WHERE dbg.gene_id=gp.id;
```

Then in R:

```
>   sql <- "SELECT * FROM db.gene AS dbg,
+           genePheno AS gp WHERE dbg.gene_id=gp.id"
>   res <- dbGetQuery(con, sql)
>   res
```

```
   gene_id _tx_id id string
1        1  72180  1   Blue
2        1  72182  1   Blue
3        2  48258  2    Red
4        2  48259  2    Red
5        9  31362  9  Green
6        9  31363  9  Green
7        9  31364  9  Green
8        9  31365  9  Green
9        9  31366  9  Green
10       9  31367  9  Green
11       9  31368  9  Green
12       9  31369  9  Green
13       9  31370  9  Green
```

The version number of R and packages loaded for generating the vignette were:

```
R version 2.15.1 (2012-06-22)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=C                 LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] hom.Hs.inp.db_2.8.0
 [2] Biostrings_2.25.3
 [3] XML_3.9-4
 [4] Homo.sapiens_1.0.0
 [5] TxDb.Hsapiens.UCSC.hg19.knownGene_2.7.1
 [6] org.Hs.eg.db_2.7.1
 [7] GO.db_2.8.0
 [8] RSQLite_0.11.1
 [9] DBI_0.2-5
[10] OrganismDbi_1.0.0
```

```
[11] GenomicFeatures_1.9.11
[12] GenomicRanges_1.9.20
[13] IRanges_1.15.10
[14] AnnotationDbi_1.20.1
[15] Biobase_2.17.5
[16] BiocGenerics_0.3.0
[17] BiocInstaller_1.8.2

loaded via a namespace (and not attached):
 [1] biomaRt_2.13.1     bitops_1.0-4.1     BSgenome_1.25.1    graph_1.35.1
 [5] RBGL_1.34.0        RCurl_1.91-1       Rsamtools_1.9.12   rtracklayer_1.17.0
 [9] stats4_2.15.1      tools_2.15.1       zlibbioc_1.3.0
```

Chapter 6

Performance: time and space

Burns [2] provides a fun and comprehensive reference for thinking about the merits and otherwise of the *R* code you write.

6.1 Measuring performance

When trying to improve performance, one wants to ensure (a) that the new code is actually faster than the previous code, and (b) both solutions arrive at the same, correct, answer.

Time The `system.time` function is a straight-forward way to measure the length of time a portion of code takes to evaluate. Here we see that the use of `apply` to calculate row sums of a matrix is much less efficient than the specialized `rowSums` function.

```
> m <- matrix(runif(200000), 20000)
> replicate(5, system.time(apply(m, 1, sum))[[1]])

[1] 0.060 0.060 0.056 0.064 0.056

> replicate(5, system.time(rowSums(m))[[1]])

[1] 0.004 0.004 0.000 0.000 0.000
```

Usually it is appropriate to replicate timings to average over vagaries of system use, and to shuffle the order in which timings of alternative algorithms are calculated to avoid artifacts such as initial memory allocation.

Comparing objects

There are many fast ways to get the wrong result – R. Gentleman

Speed is an important metric, but equivalent results are also needed. The functions `identical` and `all.equal` provide different levels of assessing equivalence, with `all.equal` providing ability to ignore some differences, e.g., in the names of vector elements.

```
> res1 <- apply(m, 1, sum)
> res2 <- rowSums(m)
> identical(res1, res2)

[1] TRUE

> identical(c(1, -1), c(x=1, y=-1))

[1] FALSE

> all.equal(c(1, -1), c(x=1, y=-1),
+           check.attributes=FALSE)

[1] TRUE
```

Profiling Two additional functions for assessing performance are `Rprof` and `tracemem`; these are mentioned only briefly here. The `Rprof` function profiles *R* code, presenting a summary of the time spent in each part of several lines of *R* code. It is useful for gaining insight into the location of performance bottlenecks when these are not readily apparent from direct inspection. Memory management, especially copying large objects, can frequently contribute to poor performance. The `tracemem` function allows one to gain insight into how *R* manages memory; insights from this kind of analysis can sometimes be useful in restructuring code into a more efficient sequence.

Exercise 18
A recent example requiring some performance tuning involved the `consensusString` function in Biostrings. This function takes a collection of aligned DNA sequences and identifies the consensus nucleotides at each site. Here is some sample data, 100 sequences each of 200,000 nucleotides.

```
> library(Biostrings)
> dna0 <- replicate(2, {
+     paste(sample(c("A", "C", "G", "T"), 200000, TRUE),
+           collapse="")
+ })
> dna <- DNAStringSet(dna0)[sample(1:2, 100, TRUE)]
```

Use `system.time` to measure how long the following function takes:

```
> system.time({
+     res0 <- consensusString(dna, ambiguityMap="?")
+ })
```

`consensusString` is an S4 method, and written in a way that makes it a little difficult to profile. Take a peak at the source code for the appropriate method. Use `system.time` and `identical` to convince yourself that `fun`, defined below, does the same thing as `consensusString`, for this particular set of data.

```
> ## like selectMethod(consensusString, "DNAStringSet")
> fun <- function(x) {
+     mat <- consensusMatrix(x, as.prob = TRUE)
+     consensusString(mat, ambiguityMap = "?", threshold = .25)
+ }
```

Use `Rprof` to determine whether time is spent in `consensusMatrix,DNAStringSet-method`, or `consensusString,matrix-method`.

As a bonus, take a peak at the slowest method and suggest some way of improving performance; perhaps there is a simple alternative for the special case that we're interested in?

As an additional bonus, investigate the performance of `consensusString` and `consensusMatrix` with different dimensions of data, e.g., many short sequences.

The microbenchmark package The `system.time` function provides one way to measure time required for function evaluation. However, time required for function evaluation often varies for reasons unrelated to implementation, e.g., load on other operating system components, garbage collection, or 'first time' costs associated with loading or allocating resources required in the function. For these reasons it is useful to replicate measures of speed, and to do so in a way that does not bias measurement toward one function or another. While one could come up with *ad hoc* approaches, the *microbenchmark* package offers a straightforward solution. The central function in this package is `microbenchmark`, with arguments being one or more functions or expressions to be evaluated coupled with simple parameters to control key features of the comparison, such as the number of times a function will be evaluated. The *microbenchmark* package is particularly useful when functions are not dramatically different in speed. A simple example

```
> library(microbenchmark)
> lst <- list(a=1:1000)
> f0 <- function(x) unlist(x)
> f1 <- function(x) unlist(x, use.names=FALSE)
> microbenchmark(f0(lst), f1(lst))
```

The default evaluates each function 100 times. The results under one system configuration show a 50-fold increase in speed associated with omitting names:

```
> microbenchmark(f0(lst), f1(lst))
Unit: microseconds
     expr      min       lq   median       uq      max
1 f0(lst) 2322.654 2331.200 2340.367 2357.686 2893.120
2 f1(lst)   42.566   44.914   49.487   56.813  100.507
```

The *rbenchmark* offers similar functionality.

6.2 Debugging

6.2.1 *R* Warnings and Errors

R signals unexpected results through warnings and errors. Warnings occur when the calculation produces an unusual result that nonetheless does not preclude further evaluation. For instance `log(-1)` results in a value `NaN` ('not a number') that allows computation to continue, but at the same time signals an warning

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Errors result when the inputs or outputs of a function are such that no further action can be taken, e.g., trying to take the square root of a character vector

```
> sqrt("two")
Error in sqrt("two") : Non-numeric argument to mathematical function
```

Warnings and errors occurring at the command prompt are usually easy to diagnose. They can be more enigmatic when occurring in a function, and exacerbated by sometimes cryptic (when read out of context) error messages.

An initial step in coming to terms with errors is to simplify the problem as much as possible, aiming for a 'reproducible' error. The reproducible error might involve a very small (even trivial) data set that immediately provokes the error. Often the process of creating a reproducible example helps to clarify what the error is, and what possible solutions might be.

Invoking `traceback()` immediately after an error occurs provides a 'stack' of the function calls that were in effect when the error occurred. This can help understand the context in which the error occurred. Knowing the context, one might use `debug` to enter into a browser (see `?browser`) that allows one to step through the function in which the error occurred.

It can sometimes be useful to use global options (see `?options`) to influence what happens when an error occurs. Two common global options are `error` and `warn`. Setting `error=recover` combines the functionality of `traceback` and `debug`, allowing the user to enter the browser at any level of the call stack in effect at the time the error occurred. Default error behavior can be restored with `options(error=NULL)`. Setting `warn=2` causes warnings to be promoted to errors. For instance, initial investigation of an error might show that the error occurs when one of the arguments to a function has value `NaN`. The error might be accompanied by a warning message that the `NaN` has been introduced, but because warnings are by default not reported immediately it is not clear where the `NaN` comes from. `warn=2` means that the warning is treated as an error, and hence can be debugged using `traceback`, `debug`, and so on.

Additional useful debugging functions include `browser`, `trace`, and `setBreakpoint`.

Fixme: tryCatch

6.3 Writing efficient scripts

6.3.1 Easy solutions

Several common performance bottlenecks often have easy solutions; these are outlined here.

Selective input Text files often contain more information, for example 1000's of individuals at millions of SNPs, when only a subset of the data is required, e.g., during algorithm development. Reading in all the data can be demanding in terms of both memory and time. A solution is to use arguments such as `colClasses` to specify the columns and their data types that are required, and to use `nrow` to limit the number of rows input. For example, the following ignores the first and fourth column, reading in only the second and third (as type `integer` and `numeric`).

```
> ## not evaluated
> colClasses <-
+   c("NULL", "integer", "numeric", "NULL")
> df <- read.table("myfile", colClasses=colClasses)
```

Recognizing 'vectorization' *R* is vectorized, so traditional programming `for` loops are often not necessary. Rather than calculating 100000 random numbers one at a time, or squaring each element of a vector, or iterating over rows and columns in a matrix to calculate row sums, invoke the single function that performs each of these operations.

```
> x <- runif(100000); x2 <- x^2
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

This often requires a change of thinking, turning the sequence of operations 'inside-out'. For instance, calculate the log of the square of each element of a vector by calculating the square of all elements, followed by the log of all elements `x2 <- x^2; x3 <- log(x2)`, or simply `logx2 <- log(x^2)`.

Pre-allocate and fill It may sometimes be natural to formulate a problem as a `for` loop, or the formulation of the problem may require that a `for` loop be used. In these circumstances the appropriate strategy is to pre-allocate the `result` object, and to fill the result in during loop iteration.

```
> ## not evaluated
> result <- numeric(nrow(df))
> for (i in seq_len(nrow(df)))
+   result[[i]] <- some_calc(df[i,])
```

Known inefficiencies Some *R* operations are helpful in general, but misleading or inefficient in particular circumstances. An example is the behavior of `unlist` when the list is named – *R* creates new names that have been made unique. This can be confusing (e.g., when Entrez gene identifiers are 'mangled' to unintentionally look like other identifiers) and expensive (when a large number of new names need to be created). Avoid creating unnecessary names, e.g.,

```
> unlist(list(a=1:2)) # name 'a' becomes 'a1', 'a2'

a1 a2
 1  2

> unlist(list(a=1:2), use.names=FALSE)   # no names

[1] 1 2
```

Names can be very useful for avoiding book-keeping errors, but are inefficient for repeated look-ups; use vectorized access or numeric indexing.

Exercise 19
Use the microbenchmark package to convince yourself of your favorite inefficiency. Can you identify inefficiencies in your own code?

Can you reason about how much copying is involved in an algorithm like pre-allocate and fill, versus say `append`*'ing to a vector while iterating over a loop?*

6.3.2 Moderate solutions

Several solutions to inefficient code require greater knowledge to implement.

Appropriate functions Using appropriate functions can greatly influence performance; it takes experience to know when an appropriate function exists. For instance, the `lm` function could be used to assess differential expression of each gene on a microarray, but the *limma* package implements this operation in a way that takes advantage of the experimental design that is common to each probe on the microarray, and does so in a very efficient manner.

```
> ## not evaluated
> library(limma) # microarray linear models
> fit <- lmFit(eSet, design)
```

Appropriate algorithms Using appropriate algorithms can have significant performance benefits, especially as data becomes larger. This solution requires moderate skills, because one has to be able to think about the complexity (e.g., expected number of operations) of an algorithm, and to identify algorithms that accomplish the same goal in fewer steps. For example, a naive way of identifying which of 100 numbers are in a set of size 10 might look at all 100×10 combinations of numbers (i.e., polynomial time), but a faster way is to create a 'hash' table of one of the set of elements and probe that for each of the other elements (i.e., linear time). The latter strategy is illustrated with

```
> x <- 1:100; s <- sample(x, 10)
> inS <- x %in% s
```

Appropriate langauge *R* is an interpreted language, and for very challenging computational problems it may be appropriate to write critical stages of an analysis in a compiled language like C or Fortran, or to use an existing programming library (e.g., the BOOST graph library) that efficiently implements advanced algorithms. *R* has a well-developed interface to C or Fortran, so it is 'easy' to do this. This places a significant burden on the person implementing the solution, requiring knowledge of two or more computer languages and of the interface between them.

Chapter 7

Using C Code

We will learn how to write C functions that can be invoked from R. This is often valuable for performance-critical algorithms that cannot be implemented efficiently in R, or when linking to existing libraries (e.g., *SAMtools*[1] to manipulate aligned sequence reads) written in C. While the latter (linking to existing C code) probably represents better justification for using C, the former (implementation and performance) probably more often motivates inclusion of C and is easier to explore in this short course.

7.1 Calling C from R

7.1.1 Example and *R* Implementation

Many algorithms can be implemented efficiently in *R*, especially when they can be implemented by using only fast *vectorized* operations (*R* and *Bioconductor* provide many), thus avoiding the use of long iterations (e.g. `for` or `lapply` loops). One class of problems that can be difficult to conceptualize in a vectorized framework involves dependence between successive elements of a vector, when the calculation of element i seems to depend on knowing the current value of element $i-1$, for instance. 'Running sum' and similar calculations fall into this category. Suppose we have a numeric vector `x` of length, e.g., `20`, and we'd like to calculate the sum of the values in windows of size `k`, e.g., if `k == 5` we'd like to compute `s[1] = sum(x[1:5])`, `s[2] = sum(x[2:6])`, . . . , `s[16] = sum(x[16:20])`.

The function `runsum0` is an *R* implementation of the 'Running sum':

```
> library(AdvancedR)
> runsum0

function (x, k)
{
    k <- as.integer(k)
    if (length(k) != 1L || is.na(k))
        stop("'k' must be a single integer")
    if (k < 1 || k > length(x))
        stop("'k' must be >= 1 and <= length(x)")
```

[1]http://samtools.sourceforge.net/

```
    end <- length(x) - k
    ans <- numeric(end + 1)
    for (i in seq_len(k)) ans <- ans + x[seq(i, end + i)]
    ans
}
<environment: namespace:AdvancedR>
```

We spend some time at the start of the function making sure inputs are valid. Then we allocate the result, a *numeric* vector of appropriate length, initialized to 0. We then iterate from 1 to k, calculating the sum in each window in a vectorized fashion. We should test this, with some easy-to-calculate values, e.g.,

```
> x <- 1:20
> stopifnot(all(x == runsum0(x, 1)))
> stopifnot(sum(x) == runsum0(x, length(x)))
> k <- 5L
> stopifnot(all(10L + k * 1:16 == runsum0(x, k)))
```

And perhaps also get a sense of how much time this implementation takes (we use the package *microbenchmark* for better timing)

```
> library(microbenchmark)
> microbenchmark(runsum0(seq_len(100000), 5),
+                runsum0(seq_len(1000000), 5),
+                runsum0(seq_len(100000), 50),
+                runsum0(seq_len(100000), 500),
+                times=5)

Unit: milliseconds
                         expr        min         lq     median         uq
1   runsum0(seq_len(1e+05), 5)   5.976092   6.592205   7.399402   7.757458
2  runsum0(seq_len(1e+05), 50)  94.346549  94.889642  95.185829  97.779747
3 runsum0(seq_len(1e+05), 500) 642.576942 702.191472 724.696595 778.973497
4   runsum0(seq_len(1e+06), 5) 203.894851 219.108547 244.185865 314.251797
        max
1  32.45118
2 257.54882
3 826.27985
4 393.34067
```

The algorithm scales approximately linearly with vector length and window size; it's useful to reflect on the algorithm and understand why that is.

However, the implementation of `runsum0` doesn't take advantage of the following observation: `s[2]` can be obtained by doing `s[1] + x[6] - x[1]`, `s[3]` by doing `s[2] + x[7] - x[2]`, etc... In a C implementation of the 'Running sum' we would of course take advantage of this in order to minimize the total number of additions/subtractions to perform.

7.1.2 The '.C' Interface

R offers two different ways to interface with C code. We'll start with the simpler `.C` interface, although as one becomes more confident it pays to move to the more comprehensive `.Call` interface.

C offers advantages in terms of speed and familiarity of programming idioms (if you know C!), but bugs can easily be introduced and the code has to be compiled. Both of these make development in C relatively slow compared to *R*, so we'd like to minimize the work that we do in C. So the function `runsum1`

```
> runsum1

function (x, k)
{
    k <- as.integer(k)
    if (length(k) != 1L || is.na(k))
        stop("'k' must be a single integer")
    if (k < 1 || k > length(x))
        stop("'k' must be >= 1 and <= length(x)")
    ans <- numeric(length(x) - k + 1)
    .C("c_runsum1", as.numeric(x), length(x), k, ans = ans)$ans
}
<environment: namespace:AdvancedR>
```

keeps the input checking in R. The `.C` interface does not allow us to allocate memory, so we also need to allocate room for the result. Note how the initial part of `runsum1` is similar to `runsum0`.

We go from *R* to C using a call to the *R* function `.C`. The function requires the name of the C-level function we want to call (in our case, `c_runsum1`) followed by arguments to the C function. In our case, we're going to pass in our input vector `x`, its length, the size of the window, and the vector we've allocated for the result.

On the other side, we've written some C code:

```
void c_runsum1(const double *x, const int *x_len, const int *k, double *ans)
{
        int i;
        int k0 = *k;
        int ans_len = *x_len - k0 + 1;

        /* initial window */
        ans[0] = 0.0;
        for (i = 0; i < k0; ++i)
                ans[0] += x[i];
        for (i = 1; i < ans_len; ++i)
                ans[i] = ans[i - 1] + x[i + k0 - 1] - x[i - 1];
}
```

The arguments to `c_runsum1` are all pointers to C basic data types, with fairly obvious mappings between their *R* equivalents, e.g., *numeric* becomes `double *`. The calling convention from *R* to C matches by position, so the fact that we named one of our arguments to `.C` `result` has no consequence for the value associated with the C function argument `result`. The return value of `c_runsum1` is `void`; we'll return a result by modifying the memory pointed to by the C `result` argument.

The remainder of the function is fairly standard C code. It relies on de-referencing the pointer arguments, remembering that while *R* indexing starts at 1, C indexing starts at 0. The actual calculation involves filling the initial window, then implementing the idea above, in 0-based vectors, that $s_i = s_{i-1} + x[i+k-1] - x[i-1]$.

Let's make a copy of the package C code in a more convenient location.

```
> c_code_dir <- system.file("c_code", package="AdvancedR")
> file.copy(c_code_dir, "~/", recursive=TRUE)
```

The code needs to be compiled into a 'shared library' before use by *R*. At the shell, evaluate the command

```
cd ~/c_code
R CMD SHLIB c_runsum.c
```

This should produce a file `~/c_code/c_runsum.so`, the shared object that we're going to use in *R*. Now, back in *R*, load the shared object

```
> dyn.load("~/c_code/c_runsum.so")
```

and test out our function

```
> runsum1(1:20, 5)
```

```
 [1] 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
```

Let's repeat our basic tests

```
> x <- 1:20
> stopifnot(all(x == runsum1(x, 1)))
> stopifnot(sum(x) == runsum1(x, length(x)))
> k <- 5L
> stopifnot(all(10L + k * 1:16 == runsum1(x, k)))
```

and check out some timings

```
> microbenchmark(runsum1(seq_len(100000), 500),
+                runsum1(seq_len(1000000), 5),
+                runsum1(seq_len(1000000), 50),
+                runsum1(seq_len(1000000), 500),
+                times=10)
```

```
Unit: milliseconds
                          expr       min        lq    median         uq
1 runsum1(seq_len(1e+05), 500)  1.461463  2.182132  2.343623   2.590227
2   runsum1(seq_len(1e+06), 5) 77.417028 84.278414 89.875946 101.094425
3  runsum1(seq_len(1e+06), 50) 78.080243 83.318569 85.911726  89.708538
4 runsum1(seq_len(1e+06), 500) 40.175053 85.527084 90.978255  98.753638
        max
1  29.59079
2 103.95190
3 101.25288
4 118.32221
```

Our algorithm appears to scale linearly with the length of `x`, and take approximately constant time in `k`. Is this as expected? How does this compare with the *R* implementation?

7.1.3 The '.Call' Interface

The `.Call` interface allows one to manipulate *R* objects at the C level. This provides quite a bit of flexibility and in the end leads to more robust code, but requires some additional work to understand and implement C code.

Here's the function we'll use for the `.Call` interface:

```
> runsum2

function (x, k)
{
    .Call("c_runsum2", as.numeric(x), as.integer(k))
}
<environment: namespace:AdvancedR>
```

The first argument is, as with `.C`, the name of the C function we'd like to invoke. The second and third arguments are the vector and window size. Notice that we expend minimal effort at the *R* level, just ensuring that `x` is a numeric vector and `k` an integer. We've moved the error checking and result creation to the C level, partly to illustrate features of *R* that are accessible with the `.Call` interface and partly because it pays to move checks and so on closer to where they are actually required.

Here is the C code at the other end of the `.Call`:

```
#include "c_runsum.h"

SEXP c_runsum2(SEXP x, SEXP k)
{
        SEXP ans;
        int x_len, k0, ans_len;
        const double *x_p;
        double *ans_p;

        /* validate inputs */
        if (!IS_NUMERIC(x))
                error("'x' must be a numeric vector");
        x_len = LENGTH(x);
        x_p = REAL(x);
        if (!IS_INTEGER(k)
         || LENGTH(k) != 1
         || (k0 = INTEGER(k)[0]) == NA_INTEGER)
                error("'k' must be a single integer");
        if (k0 < 1 || k0 > x_len)
                error("'k' must be >= 1 and <= length(x)");

        /* allocate and 'protect' ans */
[23]         ans_len = x_len - k0 + 1;
[24]         PROTECT(ans = NEW_NUMERIC(ans_len));  /* REALSXP */
[25]         ans_p = REAL(ans);
[26]
[27]         /* fill values */
[28]         c_runsum1(x_p, &x_len, INTEGER(k), ans_p);
[29]
[30]         /* 'unprotect' and return */
[31]         UNPROTECT(1);
[32]         return ans;
[33] }
```

The first line includes a header file, the `c_runsum.h` file:

```
#ifndef C_RUNSUM_H
#define C_RUNSUM_H

#include <Rinternals.h>
#include <Rdefines.h>

void c_runsum1(const double *x, const int *x_len, const int *k, double *ans);
SEXP c_runsum2(SEXP x, SEXP k);
SEXP c_rungc2(SEXP x, SEXP k);

#endif
```

which in turn includes `Rinternals.h` and `Rdefines.h`: `Rinternals.h` contains definitions of data types and the interface that we have access to (API), and `Rdefines.h` contains additional macros that add an extra level of convenience for accessing the API. The 2 files are located at:

```
> R.home("include")

[1] "/home/dtenenba/bin/R-2.15.1/include"
```

Line 3 is the signature of the C function. Our two arguments are 'S-expressions' whose type definition `SEXP` is documented in `Rinternals.h`. Rather than returning `void`, as with `.C`, we return an `SEXP` that we will allocate.

Lines 5-8 declare variables we will use in our function. We declare 3 C `int` variables, a pointer to a `const double`, and a pointer to a `double`. We declare an `SEXP` to contain the answer we will return to the user. `SEXP` is a `typedef` that is in fact a pointer to a complicated structure. For now this pointer is invalid – we have not yet allocated the structure that it will point to.

Lines 10-20 provide sanity checks on our inputs, analogous to the sanity checks in the *R* code of `runsum0`. You can see that, for the `SEXP x`, we can ask about its type (with the `IS_NUMERIC` macro) and length (with the `LENGTH` macro), and we can retrieve a pointer to the array of `double` values "contained" in the `SEXP` (with the `REAL` macro). If the `SEXP` is of type `INTSXP` (tested by `IS_INTEGER`), we access the array of `int` values with the `INTEGER` macro. We can also generate an error message, analogous to the `stop` function in *R*, with a call to `error`.

Lines 22-24 allocate the `SEXP` that will contain our answer. The `NEW_NUMERIC` macro takes the length of the numeric vector we'd like to allocate and returns that numeric vector as an `SEXP`. Note that `NEW_NUMERIC(n)` is the C equivalent of `numeric(n)` in *R*, except for the need to `PROTECT` (more on this below). This allocation is assigned to our variable `ans`. Line 25 provides us with a convenient C pointer to the array of `double` values that was allocated.

Remember in *R* that we don't explicitly manage memory – there is a 'garbage collector' that periodically looks for objects that have been allocated but are no longer in use (i.e. no longer referenced by a symbol). When we call `NEW_NUMERIC`, we request memory from *R*. Because we have not assigned this memory to an *R* symbol, we have to protect it from garbage collection. We do this via `PROTECT` – so even if some action in our C code triggers garbage collection, the memory allocated to `ans` won't be collected.

Line 28 delegates the real work of the function to the `c_runsum1` function.

Finally, lines 31-32 indicate to *R* that we no longer need protection for our `SEXP ans`, and return that `SEXP` to R. This `SEXP` is returned to the user as the result of `.Call`; if this numeric vector is assigned to an *R* variable, then it will not be garbage collected until that variable is removed or goes out of scope.

Compile and load the file with `R CMD SHLIB c_runsum.c` and `dyn.load("~/c_code/c_runsum.so")`. Let's see it in action, using the *microbenchmark* package to get a more accurate representation of timings.

```
> runsum2(1:20, 5)

 [1] 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90

> x <- 1:20
> stopifnot(all(x == runsum2(x, 1)))
> stopifnot(sum(x) == runsum2(x, length(x)))
> k <- 5L
> stopifnot(all(10L + k * 1:16 == runsum2(x, k)))
> library(microbenchmark)
> microbenchmark(runsum1_5=runsum1(seq_len(1000000), 5),
+                runsum1_50=runsum1(seq_len(1000000), 50),
+                runsum1_500=runsum1(seq_len(1000000), 500),
+                runsum2_5=runsum2(seq_len(1000000), 5),
+                runsum2_50=runsum2(seq_len(1000000), 50),
+                runsum2_500=runsum2(seq_len(1000000), 500),
+                times=10)

Unit: milliseconds
         expr       min        lq    median       uq      max
1   runsum1_5 18.719707 45.689057 49.266566 77.77654 81.09493
2  runsum1_50 20.785552 22.209091 44.407027 46.43544 83.66192
3 runsum1_500 16.587984 21.915769 44.352376 70.27538 81.71693
4   runsum2_5  8.336787 13.071693 36.039740 41.26927 44.32418
5  runsum2_50  7.449938  8.085936  9.639057 34.89732 44.48885
6 runsum2_500  7.416440 13.341771 33.303431 37.45327 42.28633
```

The performance is comparable to `.C` (differences are in milliseconds, and reflect the relatively small amount of work we do in the function; ideally we would do more replicates); the primary benefit of `.Call` is the flexibility it offers in manipulating and creating *R* objects.

As an advanced exercise, consider how you would write `rungc0` and `rungc2`, functions to determine the GC content in a sliding window of a single string (`character(1)` vector) representing a DNA sequence.

7.1.4 *Rcpp* and *inline*

There are two very interesting packages for interfacing with C and especially C++ code.

Rcpp The *Rcpp* package provides a C++ interface to *R*, masking much of the complexity of the `.Call` interface. The window example can be implemented as

```
> fl <- system.file(package="AdvancedR", "c_code", "cpp_runsum.cpp")
> noquote(readLines(fl))
```

```
 [1] #include <Rcpp.h>
 [2]
 [3] using namespace Rcpp;
 [4]
 [5] RcppExport SEXP cpp_runsum(SEXP x_in, SEXP k_in)
 [6] {
 [7]     int k, ans_len;
 [8]     NumericVector x, ans;
 [9]
[10]     try {
[11]         k = as<int>(k_in);
[12]         x = as<NumericVector>(x_in);
[13]         if (k < 1 || k > x.length())
[14]             throw not_compatible("'k' must be >= 1 and <= length(x)");
[15]     } catch (not_compatible& ex) {
[16]         forward_exception_to_r(ex);
[17]     }
[18]
[19]     ans_len = x.length() - k + 1;
[20]     ans = NumericVector(ans_len);
[21]     for (int i = 0; i < k; ++i)
[22]         ans[0] += x[i];
[23]     for (int i = 1; i < ans_len; ++i)
[24]         ans[i] = ans[i - 1] + x[i + k - 1] - x[i-1];
[25]
[26]     return wrap(ans);
[27] }
[28]
```

The include file in line 1 provides *Rcpp* headers; line 3 is a standard C++ idiom to indicate that symbols mentioned in the source fill will be searched for first in the specified C++ name space. `RcppExport` annotates the function signature to indicate external "C" linkage, to avoid C++-style name mangling.

The code illustrates several features of *Rcpp*. There are C++ classes corresponding to *R*'s `SEXP` types, e.g., `NumericVector` in line 8. The templated `as<>` serves to coerce from *R* types to C++ types. The implementation throws an error if the coercion is not possible; we can arrange to handle the error with the C++ `try / catch` (lines 10-17), or for *Rcpp* to handle the error for us. Functions like *R*'s `length()` are replaced by C++ methods (e.g., line 19). `NumericVector` and other classes have familiar subscript access to individual elements (e..g, line 22); C++ iterators and standard template library idioms can be applied to `NumericVector`. Note especially that there are no `PROTECT` statements: *Rcpp* is managing memory for us.

To compile this code into a dynamic library, we define shell environment variables that point to compiler flags that indicate where *Rcpp* header and library files are

```
cd ~/c_code
export PKG_CXXFLAGS=`R --slave -e "Rcpp:::CxxFlags()"`
export PKG_LIBS=`R --slave -e "Rcpp:::LdFlags()"`
```

compile. . .

```
R CMD SHLIB cpp_runsum.cpp
```

and back in *R* load and use the function

```
> dyn.load("~/c_code/cpp_runsum.so")
> .Call("cpp_runsum", 1:20, 5)
```

The *Rcpp* package has additional features that make it interesting to use, e.g., it is easy to incorporate *Rcpp* code into a package, to create *R* reference classes that are actually implemented in C++, and to interface to established C++ libraries. These features are discussed in the vignettes accompanying *Rcpp*

```
> vignette(package="Rcpp")
```

inline The *inline* package provides a convenient way to write code in *R* that is compiled 'on the fly' to C or C++. This is useful for quick prototyping and for development of small code chunks that might be incorporated into a package. To illustrate, we'll implement our original `.C` code. We define the signature of the function, and write the code of our original `.C` implementation as an *R character* vector:

```
> library(inline)
> sig <- signature(x ="numeric", n="integer", k="integer", result="numeric")
> code <- "
+     int i;
+     int k0 = *k;
+     int len = *n - k0 + 1;
+
+     result[0] = 0;
+     for (i = 0; i < k0; ++i)
+         result[0] += x[i];
+     for (i = 1; i < len; ++i)
+         result[i] = result[i-1] + x[i + k0 - 1] - x[i - 1];
+ "
```

We then provide these arguments to `cfunction` in the *inline* package.

```
> cfun <- cfunction(sig, code, language="C", convention=".C")
```

This actually compiles the C function – `cfun` points to a C routine ready to do our bidding:

```
> x <- 1:20
> k <- 5
> result <- numeric(length(x) - k + 1)
> cfun(x, length(x), k, result=result)$result

 [1] 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
```

Since `cfun` is compiled, it is fast.

7.2 Using C code in Packages

All C files must be placed in the `src` directory of the package. They will be automatically compiled and the resulting object files linked together into a shared object by `R CMD INSTALL`. Note that `R CMD INSTALL` also supports some advanced mechanisms to let the developer take control on how the C code will be configured and compiled via the use of a `Makeconf` or `Makefile` file, and/or a `configure` script (not covered here).

Additionally the NAMESPACE of the package needs to start with the following line:

```
useDynLib(AdvancedR)
```

Also, *R* calls to `.C` and `.Call` need to be modified to have the `PACKAGE` argument set to the name of the package. For example, in the `runsum2` function:

```
runsum2 <- function (x, k)
{
    .Call("c_runsum2", as.numeric(x), as.integer(k), PACKAGE="AdvancedR")
}
```

Finally, even though this is not strictly required, it is highly recommended to register the `.C` and `.Call` entry points. Here is the C code we use in the *AdvancedR* package for this:

```
#include "c_runsum.h"
#include <R_ext/Rdynload.h>

static const R_CMethodDef cMethods[] = {
        {"c_runsum1", (DL_FUNC) &c_runsum1, 4},
        {NULL, NULL, 0}
};

static const R_CallMethodDef callMethods[] = {
        {"c_runsum2", (DL_FUNC) &c_runsum2, 2},
        {"c_rungc2", (DL_FUNC) &c_rungc2, 2},
        {NULL, NULL, 0}
};

void R_init_AdvancedR(DllInfo *info)
{
        R_registerRoutines(info, cMethods, callMethods, NULL, NULL);
}

```

This registration mechanism is explained in details in the "5.4 Registering native routines" section of the *Writing R Extensions* manual.

7.3 Debugging

Fixme: gdb and other approaches

7.4 Embedding *R*

The essential information for embedding *R* comes from "Writing R Extensions" sections 8.1 and 8.2, and from the examples distributed with *R*. The material below covers constructing and evaluating an *R* call; dealing with the return value is a different (and in some sense easier) topic.

7.4.1 Setup

Let's suppose a Linux / Mac platform. The first thing is that *R* must have been compiled to allow linking, either to a shared or static *R* library. I work with an svn copy of *R*'s source, in the directory `~/src/R-devel`. I switch to some other directory, call it `~/bin/R-devel`, and then

```
~/src/R-devel/configure --enable-R-shlib
make -j
```

This generates `~/bin/R-devel/lib/libR.so`; perhaps whatever distribution you're using already has this? The `-j` flag runs *make* in parallel, which greatly speeds the build. Examples for embedding can be made with

```
cd ~/bin/R-devel/tests/Embedding && make
```

The source code for these examples is extremely instructive.

7.4.2 Code

The following illustrates code for embedding *R*:

```
> fl <- system.file(package="AdvancedR", "embedding", "embed.c")
> noquote(readLines(fl))
```

```
 [1] #include <Rembedded.h>
 [2] #include <Rinternals.h>
 [3]
 [4] static void doSplinesExample();
 [5]
 [6] int main(int argc, char *argv[])
 [7] {
 [8]     Rf_initEmbeddedR(argc, argv);
 [9]     doSplinesExample();
[10]     Rf_endEmbeddedR(0);
[11]     return 0;
[12] }
[13]
[14] static void doSplinesExample()
[15] {
[16]     SEXP e, result;
[17]     int errorOccurred;
[18]
[19]     // create and evaluate 'library(splines)'
[20]     PROTECT(e = lang2(install("library"), mkString("splines")));
[21]     R_tryEval(e, R_GlobalEnv, &errorOccurred);
[22]     if (errorOccurred) {
[23]         // handle error
[24]     }
[25]     // work with 'e' as in the .Call interface
[26]     UNPROTECT(1);
[27]
[28]     // 'options(FALSE)' ...
[29]     PROTECT(e = lang2(install("options"), ScalarLogical(0)));
[30]     // ... modified to 'options(example.ask=FALSE)' (this is obscure)
[31]     SET_TAG(CDR(e), install("example.ask"));
[32]     R_tryEval(e, R_GlobalEnv, NULL);
[33]     UNPROTECT(1);
[34]
[35]     // 'example("ns")'
[36]     PROTECT(e = lang2(install("example"), mkString("ns")));
[37]     R_tryEval(e, R_GlobalEnv, &errorOccurred);
[38]     UNPROTECT(1);
[39] }
```

Lines 1-4 include the headers that define the *R* embedding interface, and *R* data structures; these are located in `R.home("include")`, and serve as the primary documentation. We also have a prototype for the function that will do all the work Lines 6-12 start *R*, invoke a function that will do the work, and end *R*. The examples under the *R* directory `Embedding` include one that calls `library(splines)`, sets a named option, then runs a function `example("ns")`. This routine is repeated in lines 14-39.

7.4.3 Compile and Run

We're now ready to put everything together. The compiler needs to know where the headers and libraries are

```
g++ -I/home/user/bin/R-devel/include -L/home/user/bin/R-devel/lib -lR embed.cpp
```

The compiled application needs to be run in the correct environment, e.g., with `R_HOME` set correctly; this can be arranged easily (obviously a deployed application would want to take a more extensive approach) with

```
R CMD ./a.out
```

Depending on your ambitions, some parts of section 8 of "Writing R Extensions" are not relevant, e.g., callbacks are needed to implement a GUI on top of *R*, but not to evaluate simple code chunks.

7.4.4 Some Detail

Running through the forgoing in a bit more detail. . . An SEXP (S-expression) is a data structure fundamental to *R*'s representation of basic types (integer, logical, language calls, etc.). The line

```
PROTECT(e = lang2(install("library"), mkString("splines")));
```

makes a symbol `library` and a string "splines", and places them into a language construct consisting of two elements. This constructs an unevaluated language object, approximately equivalent to `quote(library("splines"))`. `lang2` returns an SEXP that has been allocated from R's memory pool, and it needs to be PROTECTed from garbage collection. `PROTECT` adds the address pointed to by e to a protection stack, when the memory no longer needs to be protected, the address is popped from the stack (with `UNPROTECT(1)`, a few lines down). The line

```
R_tryEval(e, R_GlobalEnv, &errorOccurred);
```

tries to evaluate `e` in *R*'s global environment. `errorOccurred` is set to non-0 if an error occurs. `R_tryEval` returns an SEXP representing the result of the function, but we ignore it here. Because we no longer need the memory allocated to store `library("splines")`, we tell R that it is no longer PROTECT'ed.

The next chunk of code is similar, evaluating `options(example.ask=FALSE)`, but the construction of the call is more complicated. The S-expression created by `lang2` is a pair list, conceptually with a node, a left pointer (CAR) and a right pointer (CDR). The left pointer of `e` points to the symbol `options`. The right pointer of `e` points to another node in the pair list, whose left pointer is `FALSE` (the right pointer is `R_NilValue`, indicating the end of the language expression). Each node of a pair list can have a TAG, the meaning of which depends on the role played by the node. Here we attach an argument name.

```
SET_TAG(CDR(e), install("example.ask"));
```

The next line evaluates the expression that we have constructed (`options(example.ask=FALSE)`), using `NULL` to indicate that we'll ignore the success or failure of the function's evaluation. A different way of constructing and evaluating this call is illustrated in `~/bin/R-devel/tests/Embedding/RParseEval.c`, adapted here as

```
PROTECT(tmp = mkString("options(example.ask=FALSE)"));
PROTECT(e = R_ParseVector(tmp, 1, &status, R_NilValue));
R_tryEval(VECTOR_ELT(e, 0), R_GlobalEnv, NULL);
UNPROTECT(2);
```

but this doesn't seem like a good strategy in general, as it mixes *R* and *C* code and does not allow computed arguments to be used in *R* functions. Instead write and manage *R* code in *R* (e.g., creating a package with functions that perform complicated series of *R* manipulations) that your *C* code uses.

The final block of code above constructs and evaluates `example("ns")`. `Rf_tryEval` returns the result of the function call, so

```
SEXP result;
PROTECT(result = Rf_tryEval(e, R_GlobalEnv, &errorOccurred));
// ...
UNPROTECT(1);
```

would capture that for subsequent processing.

7.5 Resources

Hadley Wickam's devtools c-interface provides a nice overview of `.Call`. Dirk Eddelbuettel's site provides extensive information on *Rcpp* and it's companion *Rinside* for embedding *R* inside C++.

Th 'Writing R Extensions' manual provides definitive documentation on the C interface, available with

```
> RShowDoc("R-exts")
```

Section 5, 'System and foreign language interfaces', is the place to look. Exploring `Rinternals.h` and the other header files in

```
[1] "/home/dtenenba/bin/R-2.15.1/include"
```

is also important, especially for the `.Call` interface.

Chapter 8

Parallel Evaluation

Our example involves counting reads overlapping regions of interest. The reads are from `bam` files subset to contain chromosome 4 of an RNA-seq experiment [1] using *Drosophila melanogaster*.

Here we store the locate the bam files (i.e., data) in a `BamFileList` instance from the *Rsamtools* package. We pay a little attention to naming the list elements in a way that will be convenient in subsequent steps.

```
> library(Rsamtools)
> fls <- c("treated2_chr4.bam", "treated3_chr4.bam",
+     "untreated3_chr4.bam", "untreated4_chr4.bam")
> ams <- sprintf("http://s3.amazonaws.com/AdvancedRbamfiles/%s",
+               fls)
> names(ams) <- sub(".bam$", "", basename(ams))
> ## as BamFileList
> files <- BamFileList(ams, sub(".bam$", "", ams))
```

We are interested in counting the number of reads overlapping genes in *Drosophila*. This information can be extracted from a *TxDb* package, as follows

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> features <- exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, "gene")
```

To count reads, we use `summarizeOverlaps` from the *GenomicRanges* package. This function has several different modes for counting, we'll use the default (); it's worth consulting the help page `?summarizeOverlaps` fo details. Rather than using `summarizeOverlaps` directly, we create a small wrapper that helps us use the same code in serial as well as parallel functions. The wrapper accepts an index `i` indicating the file that we are suppoed to count. The wrapper returns just the count data, whereas `summarizeOverlaps` returns more information.

```
> counter <-
+     function(i, features, files)
+     ## count overlaps for the i'th bam file (files[i])
+ {
+     se <- summarizeOverlaps(features, files[i], singleEnd=FALSE)
+     assays(se)$counts
+ }
```

In use, we have

```
> system.time({
+     res0 <- counter(1, features, files)
+ })
```

```
   user  system elapsed 
  6.465   0.000  13.469 
```

```
> head(res0, 3)
```

```
            treated2_chr4
FBgn0000003             0
FBgn0000008             0
FBgn0000014             0
```

```
> colSums(res0)
```

```
treated2_chr4 
        21802 
```

It is important to note that on a single machine, an efficient way to use `summarizeOverlaps` is simply

```
> library(parallel)
> options(mc.cores=detectCores()) ## all cores, or as appropriate
> system.time({
+         result <- summarizeOverlaps(features, files, singleEnd=FALSE)
+ })
```

8.1 *R* parallelism

To explore parallel evaluation, we write a second helper function, `doit`.

```
> doit <-
+     function(features, files, applier, FUN, ...)
+     ## apply (e.g., lapply) a FUNction to count reads in files
+     ## overlapping features, simplifying to an array
+ {
+     idx <- seq_along(files)
+     res <- applier(idx, FUN, features, files, ...)
+     res <- do.call(cbind, res)
+     dimnames(res) <- list(names(features), names(files)[idx])
+     res
+ }
```

This function takes our `features` and `files`, a function `applier` that can be used to iterate over files (e.g., `lapply` for single-processor evaluation), a function `FUN` to apply to each file (`counter`, in our case), and additional arguments that can be passed to `applier` or `FUN`. The function takes care to bind the results from each application of `FUN` together as a *matrix* with appropriate dimension names.

Here we use our `counter` and `doit` functions to count reads in two files on a single processor

```
> system.time({
+     res1 <- doit(features, files[1:2], lapply, counter)
+ })

   user  system elapsed
  8.249   0.164   8.438

> identical(res0, res1[,1, drop=FALSE])

[1] TRUE

> colSums(res1)

treated2_chr4 treated3_chr4
        21802         29250
```

On Linux or MacOS machines with with multiple cores, we've set things up to parallelize very easily. Rather than `lapply`, we can use `mclapply`, from the package *parallel* available in all recent versions of *R*. The `mclapply` function is just like `lapply`, but the 'tasks' implied by the first argument are distributed approximately evenly between the number of cores specified by the argument `mc.cores`. With `doit`, we might have

```
> library(parallel)
> system.time({
+     res2 <- doit(features, files[1:2], mclapply, counter, mc.cores=2)
+ })

   user  system elapsed
  8.212   1.736   4.943

> identical(res1, res2)

[1] TRUE
```

We have doubled our throughput and, importantly, halved the time required for evaluation. Scaling to four processors, one for each bam file, is straight-forward

```
> system.time({
+      res4 <- doit(features, files, mclapply, counter, mc.cores=4)
+ })

   user  system elapsed
 16.197   3.372   5.192

> identical(res1, res4[,1:2])

[1] TRUE

> colSums(res4)

  treated2_chr4   treated3_chr4 untreated3_chr4 untreated4_chr4
          21802           29250           29166           25042
```

With more bam files, we would choose `mc.cores` to be at most equal to the number of cores available on our machine (e.g., as reported by `detectCores()`.

`mclapply` behind the scenes The `mclapply` function is a pretty nice choice for parallel evaluation, fitting naturally with the `lapply`-like functions that are familiar to *R* programmers. `mclapply` works using the `fork` system command: the parent process creates two or more child processes. Each child is given a subset of the `X` argument to work on. The creation of child processes is relatively fast. Child processes initially have access to the same memory – same loaded packages and defined variables, for instance – as the parent process. This means that forking is very inexpensive, e.g., there is limited cost to communicate data from the parent to the child. The memory model for forked processes is 'copy-on-change', so child processes only require more memory when they modify the data they are working on; often `FUN` represents a data reduction, and it's arguments are not actually modified but instead rapidly transformed to a much smaller size.

There are trade-offs involved with parallel evaluation. A common mistake is to assign tasks that require relatively large amounts of memory. The problem is that the memory allocation of each child is amplified by the number of children – on a 16 core machine, each child requiring 8GB of additional memory would mean a total of 128GB of memory in the machine. A second issue is that many *R* functions are vectorized in the sense that some operation `f(x)` (applying a function to the vector `x`) evaluates faster than `lapply(x, f)` (applying a function to each element of `x`). The `pvec` function makes it relatively easy to take an intermediate kind of approach, dividing `x` into elements of length > 1. This represents efficient vector operations on chunks of `x`, split across several processors.

The `mclapply` function is not available on windows, and obviously does not scale above the number of cores available on a single physical computer. The *parallel* package provides functions for working across multiple machines, as well as running completely separate *R* processes on the same machine; we discuss 'socket' clusters below, a second example is clusters based on the well-established *MPI* (message passing interface) standard. There are two additional challenges with these approaches. The first is that the communication and memory footprint costs of clusters need to be given more considerations (moving big objects between processes can be very expensive). The second is that managing errors can be complicated, especially if individual tasks fail in unpredictable ways. As an anecdote, it is possible to get some amazing benefits from parallel evaluation in large clusters. In an early GWAS study here, an investigator fitting general linear models to SNPs went from a throughput of a few tens of SNPs per second with a naive implementation, to a few thousands of SNPs per second with careful code optimization on a single processor, to 100,000's of thousands of SNPs per second on a cluster with 100's of CPUs available. This transformed the problem from a batch job running over the weekend to interactive exploration of alternative models.

8.2 Clusters and clouds

Local clusters Using clusters of computers is more complicated than using processes on a single computer because the *R* session on each machine has to be set up to be similar. In addition, the costs of data transfer and the complexities of machine failure become important. To start, we create a simple cluster of independent *R* instances, running on a single machine. The `clusterEvalQ` function loads required libraries on each member of the cluster.

```
> cl <- makePSOCKcluster(4)
> libs <- clusterEvalQ(cl, {
+     library(GenomicRanges)
+     library(Rsamtools)
+ })
```

One `applier` for a socket (or other multiple-node) cluster is `parLapply`, the first argument of which is named `cl`. We perform our parallel evaluation with

```
> system.time({
+     rescl <- doit(features, files, parLapply, counter, cl=cl)
+ })

   user  system elapsed 
  0.088   0.008   6.920 

> identical(res4, rescl)
```

[1] TRUE

```
> colSums(rescl)

  treated2_chr4   treated3_chr4 untreated3_chr4 untreated4_chr4 
          21802           29250           29166           25042 
```

The Amazon cloud There are several ways to use clusters of computers from R. Here's a simple way using Amazon Web Services, Bioconductor's Amazon Machine Image (AMI), and a socket cluster from R's *parallel* package.

For the following demo, an Amazon Web Services (AWS) account is required, which in turn requires credit card information and may incur charges. You do not need to do this as an exercise; the information is provided for explanatory purposes only.

Visit the Bioconductor AMI page: http://bioconductor.org/help/bioconductor-cloud-ami/

Click "Using a parallel cluster in the cloud".

Click "Start parallel cluster".

Accept the default values. Click the IAM checkbox. Start the stack.

When the stack is running, the Outputs tab will provide a URL to *RStudio* server on the master node of the cluster. A file called `/usr/local/Rmpi/hostfile.plain`. contains the IP addresses of each machine in the cluster, and the number of cores on the machine. It might look like this:

```
10.68.155.37 4
10.50.213.89 4
10.29.191.43 4
```

Here a bit of code parses this file and constructs a string that contains each IP address multiplied by the number of cores. We then create a socket cluster using this string.

```
> library(parallel)
> tbl <- read.delim("/usr/local/Rmpi/hostfile.plain",
+                    header=FALSE, sep=" ", stringsAsFactors=FALSE)
> hosts <- rep(tbl[[1]], tbl[[2]])
> awscl <- makePSOCKcluster(hosts)
```

Looking at the cluster object, you can see that our cluster exists on three separate machines, and consists of 12 cores altogether.

```
> awscl

socket cluster with 12 nodes on hosts '10.190.38.61', '10.6.155.113', '10.159.30.223'
```

Now we run a trivial function on the cluster:

```
> system.time(res <- clusterCall(awscl, Sys.sleep, 1))

user  system elapsed
0.004   0.000   1.005
```

The output shows that the function ran in parallel (i.e, took 1 second, not 12).

Overlapping read counting, revisited Now we run doit on our 12-node cluster. But if we think about it, we don't really need 12 nodes. There are only 4 BAM files to process. So we could have just started a 4-node cluster (or done all processing on a single 4-core machine, as we have already demonstrated). Instead, we'll subset the 12-node cluster and end up with one that just has four nodes.

```
> hosts[c(1, 2, 5, 9)]

[1] "10.190.38.61"  "10.190.38.61"  "10.6.155.113"  "10.159.30.223"

> bamcl = awscl[c(1, 2, 5, 9)]
> bamcl

socket cluster with 4 nodes on hosts '10.190.38.61', '10.6.155.113', '10.159.30.223'
```

We need to prepare our new cluster by telling it to load the packages we'll need:

```
> libs <- clusterEvalQ(bamcl, {
+     library(GenomicRanges)
+     library(Rsamtools)
+ })
```

Now we're ready to run doit() again.

```
> system.time({
+     rescl <- doit(features, files, parLapply, counter, cl=bamcl)
+ })

   user  system elapsed
  0.708   0.104  23.691
```

Note: When using Amazon Web Services, be sure and turn off resources when you are done with them! Otherwise, charges will continue to accrue. To stop the cluster we've started, go back to the CloudFormation Management Console page, select the stack we started, click Delete Stack, and confirm deletion.

8.3 C parallelism

There are relatively few examples of C-level parallelism in *R*. One reason is because *R*'s C entry points are generally not thread-safe – two independent threads cannot call in to *R* simultaneously. Nonetheless, *R* does provide support for use of the *OpenMP* parallel programming specification. *OpenMP* allows programmers to define `pragma`'s that indicate to an *OpenMP*-aware compiler that the code can be compiled to allow for parallel evaluation. It is the programmer's responsibility to ensure that the code is safe to be evaluated in parallel, and that the use of `pragma`s is actually effective at increasing speed (this can be surprisingly challenging to achieve). One subtle advantage of C parallelism is that the user will almost certainly be unaware of the implementation details – their *R* code will appear to just 'run faster'.

Here is a snippet from the *ShortRead* package, where a buffer containing many fastq records is being parsed for geometry. The `for` loop operates independently on each read, and there are no function calls to compromise thread safety.

```
  /* geometry */
#ifdef SUPPORT_OPENMP
#pragma omp parallel for
#endif
  for (int i = 0; i < fastq->n_curr; ++i) {
    const Rbyte *buf = fastq->records[i].record;
    const Rbyte *start;

    start = ++buf;           /* id; skip '@' */
    while (*buf != '\n')
       ++buf;
    id_w[i] = buf - start;
    while (*buf == '\n')
      ++buf;
    sread_w[i] = 0;          /* read */
    while (*buf != '+') {
      while (*buf++ != '\n') /* strip '\n' */
        sread_w[i] += 1;
    }
  }
```

The code uses a macro `SUPPORT_OPENMP` determined by *R* when *R* was installed. If the macros is defined, then an openMP directive is inserted that tell the compiler to parallelize the following `for` loop. Some basic considerations point to the challenges of effectively parallizing C code. For instance, Amdahl's law points out that if a fraction P of the code is parallelized across N threads, the overall code speed-up is at maximum $1/((1-P)+P/N)$ – even if were an infinite number of threads, the code only runs $1/(1-P)$-fold faster. So if only a small fraction of our C code, which in turn is only a small fraction of our overall code, can be parallelized, our efforts at parallizing C code may not give us much in overall performance.

Chapter 9

An Extended Example

9.1 Package tour

9.1.1 *Bioconductor* packages

A brief, slightly dated, summary of *Bioconductor* packages available for sequence analysis is presented in Table 9.1; see the BiocViews[1] section of the web site for a current listing.

9.1.2 Common work flows

Manipulating reads, counting overlaps

Differential representation

Annotation

Annotation of called variants

9.2 Highlights

The following highlight best practices or other interesting features in *Bioconductor* packages produced by our group. Not all packages adopt all approaches.

9.2.1 Package structure

File structure Similar organization of files in `R`, `man`, `inst/unitTests` directories.

DESCRIPTION Explicit collation order. Version dependencies. Appropriate use of Depends:, Imports:, Suggests: fields.

[1] http://bioconductor.org/packages/release/BiocViews.html#___Software

Table 9.1: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *GenomicFeatures*, *Biostrings*, *BSgenome*, *girafe*.
Input / output	*ShortRead* (fastq), *Rsamtools* (bam), *rtracklayer* (gff, wig, bed), *VariantAnnotation* (vcf), *R453Plus1Toolbox* (454).
Annotation	*GenomicFeatures*, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*gmapR*, *Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *Rolexa*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Motifs	*BCRANK*, *cosmo*, *cosmoGUI*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Copy number	*cn.mops*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*ArrayExpressHTS*, *Genominator*, *easyRNASeq*, *oneChannelGUI*, *rnaSeqMap*.
Database	*SRAdb*.

NAMESPACE Shared object use via `useDynLib`. Tightly controlled imports, explicit exports. Reuse of generics and classes.

Unit tests Moderate use, especially in more recent development cycles. Often introduced in response to bug reports.

Help pages Working examples. Cross-references. Standardized (?) content.

Vignettes Extensive vignettes with working code. Challenge: vignettes often written when the package is originally developed, and do not track the leading edge of package development.

9.2.2 Classes and methods

Class hierarchy Extensive class hierarchy. Challenge: hard to avoid overwhelming users; some classes were useful once (e.g., the *AlignedRead* class in the *ShortRead* package) but would not be a 'first choice' (*GappedAlignments* in *GenomicRanges*) now.

Performance Approaches to minimizing the number of S4 objects, e.g., in *GRangesList* instances.

9.2.3 Data resources

Use of sqlite

Retrieval via *rtracklayer*

9.2.4 C code

Important implementations Examples: *XString* and related classes. `findOverlaps`. Run-length encoding.

Reuse of third-party code Examples: *Rsamtools*, overlap code in *IRanges*, UCSC access in *rtracklayer*.

Registration Different approaches, e.g.: *IRanges* registration at C level; *Rsamtools* registration in name space.

9.2.5 ...

References

[1] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, pages 193–202, 2011.

[2] P. Burns. The R inferno. Technical report, 2011.

[3] J. M. Chambers. *Software for Data Analysis: Programming with R.* Springer, New York, 2008.

[4] P. Dalgaard. *Introductory Statistics with R.* Springer, 2nd edition, 2008.

[5] R. Gentleman. *R Programming for Bioinformatics.* Computer Science & Data Analysis. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[6] R. Kabacoff. *R in Action.* Manning, 2010.

[7] N. Matloff. *The Art of R Programming.* No Starch Pess, 2011.